Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework

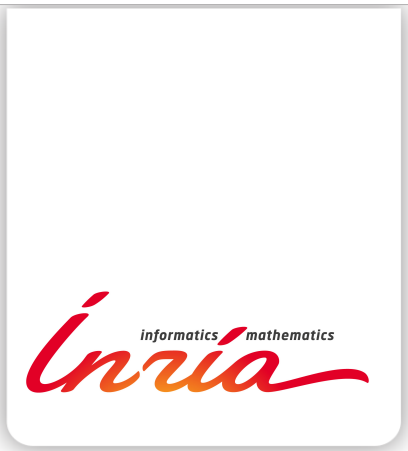
Gurvan Le Guernic, José Angel Galindo Duarte

Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework

Gurvan LE GUERNIC, José A. GALINDO

RESEARCH REPORT
N° 8967
October 2016
Project-Teams DiverSE

ISSN 0249-6399 ISRN INRIA/RR--8967--FR+ENG



Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework

Gurvan Le Guernic, José A. Galindo

Project-Teams DiverSE

Research Report n° 8967 — October 2016 — 38 pages

Abstract: Many project-specific languages, including in particular filtering languages, are defined using non-formal specifications written in natural languages. This leads to ambiguities and errors in the specification of those languages. This paper reports on an experiment on using a tool-supported language specification framework (𝕂) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. In the context of this experimentation, the cost and benefits of formally specifying a language using a tool-supported framework in general (as well as the expressivity and ease of use of the 𝕂 framework in particular) are evaluated.

Key-words: Formal specification, Language, Semantics, Packet filtering, K framework

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Retour d'expérience sur l'utilisation du framework K pour la spécification formelle d'un language de filtrage de trames

Résumé : De nombreux langages spécifiques à un projet, entre autre les langages de filtrage, sont définis dans une spécification non-formelle écrite en langage naturel. Ces spécifications sont par conséquence souvent ambiguës et erronées. Ce rapport est un retour d'expérience sur l'utilisation d'un framework outillé de spécification de langage (le framework K) pour la spécification formelle de la syntaxe et sémantique d'un langage de filtrage de trames ayant une complexité similaire à celle rencontrée sur des projets réels. Dans le contexte de cette expérimentation, ce rapport évalue les coûts et bénéfices liés à une démarche de spécification formelle d'un langage en s'appuyant sur un framework outillé en général, et plus particulièrement dans le cas du framework K.

Mots-clés : Spécification formelle, Langage, Sémantique, Filtrage de trames, Framework K

1 Introduction

Packet filtering (accepting, rejecting, modifying or generating packets, i.e. strings of bits, belonging to a sequence) is a recurring problematic in the domain of information systems security. Such filters can serve, among other uses, to reduce the attack surface by limiting the capacities of a communication link to the legitimate needs of the system it belongs to. This type of filtering can be applied to network links (which is the most common use), product interfaces, or even on the communication buses of a product. If the filtering policy needs to be adapted during the deployment or operational phases of the system or product, it is often required to design a specific language $\mathcal{L}$ (syntax and semantics) to express new filtering policies during the lifetime of the system or product. This language is the basis of the filters that are applied to the system or product. Hence, it plays an important role in the security of this system or product. It is therefore important to have strong guarantees regarding the expressivity, precision, and correction of the language $\mathcal{L}$. Those guarantees can be partly provided by a formal design (and development) process.

Among diverse duties, the DGA (Direction Générale de l'Armement, a french procurement agency) is involved in the supervision of the design and development of filtering components or products. Those filters come in varying shapes and roles. Some of them are network apparatuses filtering standard Internet protocol packets (such as firewalls); while others are small parts of integrated circuits filtering specific proprietary packets transiting on computer buses. Their common definition is: "a tool sitting on a communication channel, analyzing the sequence of packets (strings of bits with a beginning and an end) transiting on that channel, and potentially dropping, modifying or adding packets in that sequence". Whenever the filtering algorithm applied is fixed for the lifetime of the component or product, this algorithm is often "hard coded" into the component or product with the potential addition of a configuration file allowing to slightly alter the behavior of the filter. However, sometimes the filtering algorithm to apply may depend on the deployment context, and may have to evolve during the lifetime of the component or product to adapt to new uses or attackers. In this case, it is often necessary to be able to easily write new filtering algorithms for the specific product and context. Those algorithms are then often described using a Domain Specific Language (DSL) that is designed for the expression of a specific type of filters for a specific product. The definition of the syntax and semantics of this DSL is an important task. This DSL is the link between the filtering objectives and the process that is really applied on the packet sequences. The DSL used must be expressive enough to describe the desired filtering algorithm and precise enough to avoid mismatches between the intention and the realization, while being simple to use.

This paper is an experience report on the use of a tool-supported language specification framework (the $\mathbb{K}$ framework) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. The tool used to formally specify the DSL is introduced in Sect. 2. For confidentiality reasons, in order to be allowed by the DGA to communicate on this experimentation, the language specified for this experiment is not linked to any particular product or component. It is a generic packet filtering language that tries to cover the majority of features required by packet filtering languages. This language is introduced in Sect. 3 while its formal specification is described in Sect. 4. This language is tested in Sect. 5 by implementing and simulating a filtering policy enforcing a sequential interaction for a made-up protocol similar to DHCP. Before concluding in Sect. 7, this paper discusses the results of the experimentation in Sect. 6.

2 Introduction to the $\mathbb{K}$ Framework

Surprisingly, even if it is a niche for tools, there exists quite a number of tools specifically dedicated to the formal *specification* of languages (our focus in this work is on specifying rather than implementing DSLs). Those tools include among others: PLT Redex [6, 12], Ott [21], Lem [17], Maude MSOS Tool [3], and the $\mathbb{K}$ framework [18, 24]. All those tools focus on the (clear formal) specification of languages rather than their (efficient) implementation, which is more the focus of tools and languages such as Rascal [15, 2, 14] or its ancestor The Meta-Environment [13, 23], Kermeta [8, 9], and others. PLT Redex is based on reduction relations. PLT Redex is an extension (internal DSL) of the Racket programming language. Ott and Lem are more oriented towards theorem provers. Ott and Lem allow to generate formal definitions of the language specified for Coq, HOL, and Isabelle. In addition, Lem can generate executable OCaml code. Ott is more programming language syntax oriented, while Lem is a more general purpose semantics specification tool. Ott and Lem can be used together in some contexts. The Maude MSOS Tool, whose development has stopped in 2011, is based on an encoding of modular structural operational semantics (MSOS) rules into Maude. Similarly to the Maude MSOS Tool, the $\mathbb{K}$ framework is based on rewriting and was also originally implemented on top of Maude. Its implementation is now moving to Java.

The goal set for the experiment reported in this paper is to evaluate the usability of an "appropriate" tool for the "formal" specification of a packet filtering language by an "average" engineer. The "appropriate" tool needs then to: be easy to use; be able to produce (or take as input) "human readable" language specifications; provide some "formal" correctness guarantees; and be executable (simulatable) in order to test (evaluate) the language specified. The $\mathbb{K}$ framework seems to meet those requirements and has been chosen to be the "appropriate" tool after a short review of available tools. There is no claim in this paper that the $\mathbb{K}$ framework is better than the other tools, even in our specific setting.

This section introduces the $\mathbb{K}$ framework [19] by relying on the example of a language allowing to compute additions over numbers using Peano's encoding [7]. The $\mathbb{K}$ source code of this language specification is provided in Fig. 1. A $\mathbb{K}$ definition is divided into three parts: the *syntax* definition, the *configuration* definition, and the *semantics* (rewriting rules) definition. The definition of the language *syntax* is given in a module whose name is suffixed with "-SYNTAX". It uses a BNF-like notation [1, 16]. Every non-terminal is introduced by a `syntax` rule. For example, the definition of the notation for numbers (`Nb`) in this language, provided on line 2 of Fig. 1, is equivalent to the definition given by the regular expression "(`Succ`)* `Zero`".

The *configuration* definition part is introduced by the keyword `configuration` and defines a set of (potentially nested) cells described in an XML-like syntax. This configuration describes the "abstract machine" used for defining the semantics of the language. The initial state (or configuration) of the abstract machine is the one described in this configuration part. The parsed program (using the syntax definition of the previous part) is put in the cell containing the `$PGM` variable (of type `K`). For the `Peano` language, the `env` cell is used to store variable values in a map initially empty (`.Map` is the empty map). From this definition, the $\mathbb{K}$ framework can produce a graphical representation of the configuration, provided in Fig. 2

Figure 2: Peano's $\mathbb{K}$ configuration

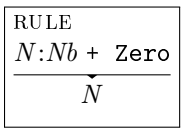
The *semantics* definition part is composed of a set of rewriting rules, each one of them introduced by the keyword `rule`. In the $\mathbb{K}$ source file, rules are roughly denoted as "CCF `=>` NCF" where CCF and NCF are configuration fragments. The meaning of "CCF `=>` NCF" can be summarized as: if CCF is a fragment of the current abstract machine state (or configuration)

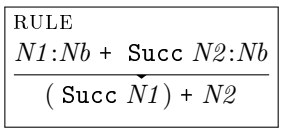
```
1  module PEANO-SYNTAX
     syntax Nb ::= "Zero" | "Succ" Nb
3    syntax Exp ::= Nb | Id | Exp "+" Exp [strict,left]
     syntax Stmt ::= Id ":=" Exp ";" [strict(2)]
5    syntax Prg ::= Stmt | Stmt Prg
   endmodule
7
   module PEANO imports PEANO-SYNTAX
9    syntax KResult ::= Nb

11   configuration
       <env color="green"> .Map </env>
13     <k color="cyan"> $PGM:K </k>

15   rule N:Nb + Zero => N
     rule N1:Nb + Succ N2:Nb => ( Succ N1 ) + N2
17
     rule
19     <env> ... Var:Id |-> Val:Nb ... </env>
       <k> ( Var:Id => Val:Nb ) ... </k>
21
     rule
23     <env> Rho:Map (.Map => Var |-> Val ) </env>
       <k> Var:Id := Val:Nb ; => . ... </k>
25     when notBool (Var in keys(Rho))

27   rule
       <env> ... Var |-> ( _ => Val ) ... </env>
29     <k> Var:Id := Val:Nb ; => . ... </k>

31   rule S:Stmt P:Prg => S ~> P [structural]
   endmodule
```

Figure 1: $\mathbb{K}$ source file of the Peano example

then the rule may apply and the fragment matching CCF in the current configuration would then be replaced by the new configuration fragment NCF. In order to increase the expressivity of rules, CCF may contain free variables that are reused in expressions in NCF. If a specific valuation of the free variables V in CCF allows a fragment of the current configuration to match CCF, then this fragment may be replaced by NCF where the variables V are replaced by their matching valuation.

The rules for addition over numbers (`Nb` and not `Exp`), on lines 15 and 16 of Fig. 1, follows closely this representation. For those rules, CCF is a program fragment that can be matched in any cell of the configuration. For those two rules, the $\mathbb{K}$ framework can then produce the following graphical representations:





For other rules, the configuration fragment matching is more complex and involves precise configuration cells that are explicitly identified. In order to compress the representation, CCF and NCF are not stated separately anymore. The common parts are stated only once, and the parts differing are again denoted "CCF_i => NCF_i", where CCF_i is a sub-fragment in CCF and NCF_i is the corresponding sub-fragment in NCF. Cells that have no impact on a rule R and

are not impacted by R do not appear explicitly in the rule. Cells heads and tails (potentially empty) that are not modified by a rule can be denoted "`...`", instead of using a free variable that would not be reused.

For example, the rule which starts on line 18 of Fig. 1 is the rule used to evaluate variables. The current configuration needs to contain a mapping from a variable `Var` to a value `Val` ("`X |-> V`" denotes a mapping from `X` to `V`) somewhere in the map contained in the `env` cell. It also needs to contain the variable `Var` at the beginning of cell `k`. This rule has the effect of replacing the instance of `Var` at the beginning of cell `k` by the value `Val`. For this rule, the $\mathbb{K}$ framework generates the graphical representation given in Fig. 3.

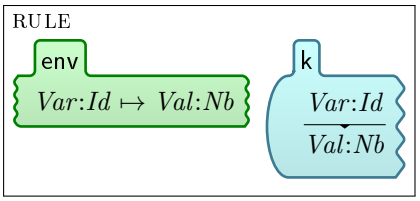


Figure 3: Peano's $\mathbb{K}$ rule for variables

The last rule on line 31 involves other internal aspects of the $\mathbb{K}$ framework. It roughly states that, in order to evaluate a statement `S` followed by the rest `P` of the program, `S` must first be evaluated to a `KResult` (defined on line 9) and then `P` is evaluated.

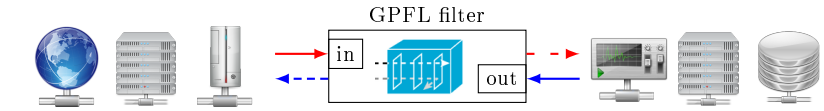
3 GPFL Context

The language specified in the experiment reported in this paper, named GPFL, is a generic packet filtering language. For obvious confidentiality reasons, GPFL is not a language actually used in any specific real product. GPFL has been made-up in order to be able to communicate on the experimentation on tool supported formal specification of filtering languages reported in this paper. However, GPFL covers the majority of features needed in packet filtering languages dealt with by the DGA. GPFL can be seen as the "mother" of the majority of packet filtering languages.

GPFL aims at expressing a wide variety of filters. Those filters can be placed at the level of network, interfaces, or even communication buses between electronic components. They can be applied on standard protocols such as IP, TCP, UDP, ... or on proprietary protocols, which are more common for component communication protocols. However, all those filters are assumed to be placed on a communication link. Messages (packets) that get through the filter can only get through in two ways, either "going in" or "going out"; there is no switching taking place in GPFL filters. Those different use cases are illustrated in Fig. 4.

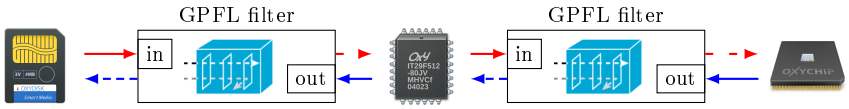
GPFL focuses on the internal logic of the filter. Decoding and encoding of packets is assumed to be handled outside of GPFL programs (filters), potentially using technologies such as ASN.1 [10, 5]. For GPFL programs, a packet is a record (a set of valued fields). A GPFL program (dynamically) inputs a sequence of records and outputs a sequence of records. Figure 5 describes the architecture of GPFL-based filters. An incoming packet (on either side) is first parsed (decoded) before being handed over to the GPFL program. If the packet can not be parsed, depending on the type of filter (white list or black list), the packet is either dropped or passed to the other side without going through the GPFL program. Any packet (record) output by the GPFL program (on either side) is encoded before being sent out. In addition, the GPFL program can generate alarms due to packets not complying with the encoded filtering policy.

The GPFL language must allow to: drop, modify or accept the current packet being filtered; generate new packets; and generate alarms. GPFL must allow to base the decision to take any of those actions on information pieces concerning the current packet being filtered and previously filtered packets. Those information pieces must include: some timing information, current or previous packets directions through the filter ("in" or "out"), and characteristics of current or



(a) Network filtering

(b) Interface filtering



(c) Bus filtering

Figure 4: Use cases for GPFL-based filters

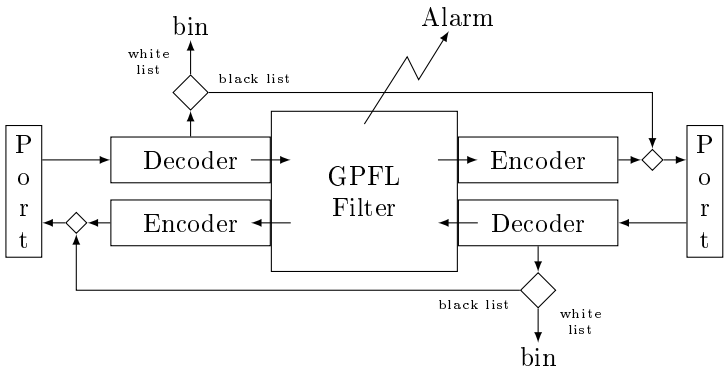


Figure 5: Architecture of GPFL-based filters

previous packets including field values and computed properties such as, for example, a packet "type" or total length. The computation of those properties and decoding of packet fields is outside of the scope of GPFL; it is left to the decoders.

In order to gradually build a decision, GPFL must allow to interact with variables (reading, writing, and computing expressions) and automata (triggering a transition in an automaton and querying its current state). The intent for automata is to be used to track the current step of sessions of complex protocols. GPFL must allow to combine filtering statements using: sequential control statements (executing two statements in sequence); conditional control statements (executing a statement only if a condition is true); iterating control statements (repeatedly executing a statement for a fixed number of repetitions). There is no requirement for a loop (or while) statement whose exit condition is controlled by an expression recomputed after every iteration. For the experiment reported in this paper (on formal specification of a filtering language), the iterating statement is considered sufficient for the intended use of GPFL and close enough to a loop statement from a semantics point of view, while exhibiting interesting properties for future

analyses (for example, any GPFL program terminates).

4 Formal Specification of GPFL

4.1 Syntax

The syntax of GPFL is formally defined by the $\mathbb{K}$ source fragment provided in Fig. 6. A GPFL

```
syntax ExpVal ::= Int | Bool | String | AEvtId | Port

syntax BuiltInId ::= "_inPort"
syntax VarId ::= Id
syntax FieldId ::= "$" Id
syntax AutomatonId ::= "#" Id
syntax ExpId ::= BuiltInId | VarId | FieldId | AutomatonId

syntax UnaryOp ::= "--" | "!"
syntax BinaryOp ::= "+" | "-" | "*" | "/" | "&" | "|"
                  | "==" | "<" | ">" | "<=" | ">="

syntax Exp ::= ExpVal | ExpId
             | UnaryOp Exp            [strict(2)]
             | Exp BinaryOp Exp       [strict(1,3), left]
             | "(" Exp ")"            [bracket]

syntax Cmd ::= "nop" | "accept" | "drop" | "send(" Port "," Fields ")"
             | "alarm(" Exp ")"                                  [strict(1)]
             | "set(" Id "," Exp ")"                             [strict(2)]
             | "newAutomaton(" String "," AutomatonId ")"
             | "step(" AutomatonId "," Exp "," Stmt ")"          [strict(2)]
syntax Stmt ::= Cmd
              | "cond(" Exp "," Stmt ")"                         [strict(1)]
              | "iter(" Exp "," Stmt ")"                         [strict(1)]
              | "newInterrupt(" Int "," Bool "," Stmt ")"
              | Stmt Stmt                                        [right]
              | "{" Stmt "}"                                     [bracket]

syntax AutomataDef ::= "AUTOMATA" String AutomataDefTail
syntax AutomataDefTail ::= "init" "=" AStateId ATransitions | ATransitions
syntax ATransitions ::= List{ATransition,""}
syntax ATransition ::= AStateId "-" AEvtId "->" AStateId
syntax AStateId ::= String
syntax AEvtId ::= String
syntax InitSeq ::= "INIT" Stmt
syntax PrologElt ::= AutomataDef | InitSeq
syntax Prologues ::= PrologElt | PrologElt Prologues

syntax Program ::= "PROLOGUE" Prologues "FILTER" Stmt
```

Figure 6: $\mathbb{K}$ source file of GPFL syntax

program is composed of a prologue, executed only once in order to initialize the execution environment, and a filter statement, executed once for every incoming packet.

A prologue is composed of automaton kind definitions and initialization sequences. An automaton kind definition specifies an identifier K, an initial state for automata of kind K and a set of transitions for automata of kind K. A transition definition is composed of: two automaton states F and T, and an automaton event that triggers the transition from F to T.

A GPFL statement is composed of GPFL commands or statements combined sequentially. Some statements can be guarded by an expression and executed only if that expression evaluates to true (`cond`). Some statements, associated with an expression e, can be executed multiple times (`iter`), as much times as the expression e evaluates to before the first iteration. Finally, the `newInterrupt` statement registers a statement to be executed in the future, potentially periodically.

GPFL commands are the basic units having an effect on the execution environment. The `nop` command has no effect and serves mainly as a place holder. The `accept`, resp. `drop`, command states to accept, resp. drop, the current packet and stop the filtering process for this packet. The `send` command sends a packet on one of the ports. The `alarm` command generates a message on the alarm channel. The `set` command sets the value of a variable. The `newAutomaton` command initializes an automaton of the provided kind and assigns the provided identifier to interact with this newly created automaton. The `step` command tries to trigger an automaton transition by sending an event e to an automaton a. If there is no transition from the current state of a triggered by the event e, then the associated statement is executed.

Expressions in GPFL are quite standard. Primitive values include integers, booleans, strings, automata events and ports. The only "somewhat" uncommon aspect of GPFL is that automaton identifiers in expressions are evaluated to the current state of the associated automaton.

4.2 Configuration

The configuration used to execute GPFL programs is presented graphically in Fig. 7. A configuration contains a set of automaton kind definitions (`automatonDef`), with the same information as defined in Sect. 4.1. The `prg` cell contains the GPFL program. After initialization of the program, the `filter` cell contains the filter (GPFL statement) that is to be executed for every packet. The `env` cell is the main dynamic part of the execution environment. It corresponds to a "record" of maps that associate: automaton kind and current state to automaton identifiers (`automata` cell), and values to variables.

The only time related feature available to GPFL execution machinery (in addition to packet arrival time) are interrupts. The configuration contains an `interrupts` cell. This cell contains a set of interrupt definitions (`interrupt*`). An interrupt is a triplet composed of: the time when the interrupt is to be triggered, the code to be executed, and a "Time" value equal to the interruption period for a periodic interruption or nothing for a non-periodic interruption. In addition, the `interrupts` cell contains an ordered list of the next "times" when an interrupt is to be executed.

The `input` cell contains the current packet to be filtered, with its arrival time and port. The configuration also contains a `k` cell that holds the GPFL statement under execution. Each time a new packet is input, the content of the `k` cell is replaced by the content of the `filter` cell.

Finally, the `streams` cell contains: the packet input stream divided into the next packet to arrive (`inHead`) and the rest of the stream (`inTail`), the packet output stream, and the alarm output stream. In the input stream, resp. output stream, packets arriving, resp. leaving, on both ports are mixed together, but contains information on the port of entry, resp. exit. Some choices made to represent those streams are not an intrinsic part of the formal specification of GPFL. The division of the input stream into a head and a tail is such a choice. Those choices are made in order to be able to execute the specification. It is then required to implement, in the $\mathbb{K}$ framework, a mechanism to retrieve and parse strings describing packet sequences sent to the filter. In order to help distinguish between the formal specification of GPFL and the mechanisms put in place to execute it, whenever possible, implementation choices, such as the format of strings describing packets, are defined in another file which is loaded with the `require`

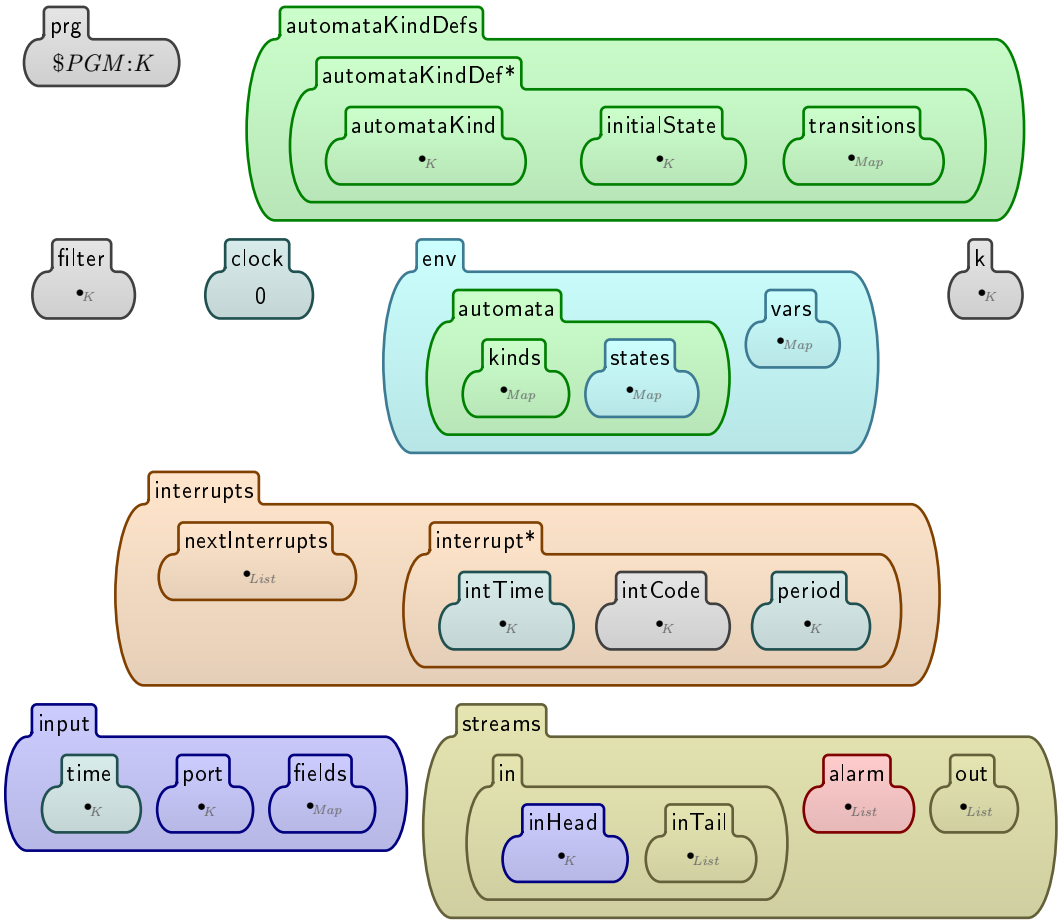


Figure 7: $\mathbb{K}$ configuration of GPFL

instruction.

4.3 Semantics

The formal specification of GPFL's semantics relies on two auxiliary specifications. The first one define specific data types and associated functions (Fig. 8). The second one defines auxiliary conversion functions between those data types and `String` (Fig. 9).

The formal specification of GPFL's semantics includes the usual rules for handling expressions that can be found in many $\mathbb{K}$ examples or tutorials. The `strict` attributes of the syntax rules on lines 22 and 23 of Fig. 6 specify that operation arguments in expressions have to be evaluated to values first. The rules in Fig. 10 specify the semantics of variables, which consists simply in retrieving their values in the corresponding configuration cell. The $\mathbb{K}$ source provided in Fig. 11 specifies the semantics of operations applied to values.

The rest of GPFL's semantics is decomposed in three execution phases: **(a)** the program initialization, **(b)** the selection of the next statement to execute, and **(c)** the execution of the selected statement. Phase (a) occurs only once at the beginning of the execution; then phases (b) and (c) are repeatedly executed one after the other. Phase (b) selects the statement associated

```
module GPFPL-DATA

  syntax Time ::= Int

  syntax Time ::= Time "+Time" Time [function]
  rule T1:Int +Time T2:Int => T1 +Int T2 [structural]

  syntax Bool ::= Time "<Time" Time [function]
  rule T1:Int <Time T2:Int => T1 <Int T2 [structural]

  syntax Port ::= "inSide" | "outSide"

  syntax Bool ::= Port "==Port" Port [function]
  rule P1:Port ==Port P2:Port => P1 ==K P2 [structural]

  syntax Port ::= "oppositePort(" Port ")" [function]
  rule oppositePort( inSide:Port ) => outSide
  rule oppositePort( outSide:Port ) => inSide

  syntax Fields ::= Map

  syntax Bool ::= Id "in" Fields [function]
  rule X:Id in MF:Map => (X in keys(MF)) [structural]

  syntax K ::= Fields ".getValueOfField(" Id ")" [function]
  rule MF:Map .getValueOfField( X:Id )  => MF[X] [structural]

  syntax PktDescr ::= "(" Time "," Port "," Fields ")"

  syntax Time ::= "getTimeFromPkt(" PktDescr ")" [function]
  rule getTimeFromPkt( ( T:Time , _:Port , _:Fields ) ) => T

  syntax Port ::= "getPortFromPkt(" PktDescr ")" [function]
  rule getPortFromPkt( ( _:Time , P:Port , _:Fields ) ) => P

  syntax Fields ::= "getFieldsFromPkt(" PktDescr ")" [function]
  rule getFieldsFromPkt( ( _:Time , _:Port , MF:Fields ) ) => MF

endmodule
```

Figure 8: $\mathbb{K}$ source file of specific data types

```
require "dataDefs.k3"

module STRING-CONVERSIONS

  imports GPFPL-DATA

  syntax TimeStr ::= Int

  syntax String ::= "time2Str(" Time ")" [function]
  rule time2Str( T:Int ) => Int2String( T )
  syntax Time ::= "str2Time(" TimeStr ")" [function]
  rule str2Time( T:Int ) => T

  syntax PortStr ::= Port

  syntax String ::= "port2Str(" Port ")" [function]
  rule port2Str( inSide ) => "inSide"
  rule port2Str( outSide ) => "outSide"
  syntax Port ::= "str2Port(" PortStr ")" [function]
  rule str2Port( P:Port ) => P

  syntax FieldStr ::= Id "=" String
  syntax FieldsStr ::= List{ FieldStr , "," }

  syntax String ::= "fields2Str(" Fields ")" [function]
  rule fields2Str( .Map ) => ""
  rule fields2Str( F:Id |-> V:String ) => ( Id2String(F) +String "=" +String
    "\"" +String V +String "\"" )
  rule fields2Str( F:Id |-> V:String FTail:Map ) => ( fields2Str(F |-> V) +
    String "," +String fields2Str(FTail) )

  syntax Fields ::= "str2Fields(" FieldsStr ")" [function]
  rule str2Fields( M:FieldsStr ) => str2mfInternals(M)
  syntax Map ::= "str2mfInternals(" FieldsStr ")" [function]
  rule str2mfInternals( .:FieldsStr ) => .Map
  rule str2mfInternals( F:Id = V:String ) => (F |-> V)
  rule str2mfInternals( F:Id = V:String , MFS:FieldsStr ) => (F |-> V)
    str2mfInternals(MFS)

  syntax PktStr ::= "(" TimeStr ";" PortStr ";" FieldsStr ")"

  syntax String ::= "pkt2Str(" Time "," Port "," Fields ")" [function]
  rule pkt2Str( T:Time , P:Port , M:Map ) => ( "(" +String time2Str(T) +
    String "; " +String port2Str(P) +String "; " +String fields2Str(M) +
    String ")" )

  syntax PktDescr ::= "pktStr2pktDescr(" PktStr ")" [function]
  rule pktStr2pktDescr( ( T:TimeStr ; P:PortStr ; MF:FieldsStr ) ) => (
    str2Time(T) , str2Port(P) , str2Fields(MF) )

endmodule
```

Figure 9: $\mathbb{K}$ source file of `String` conversion functions

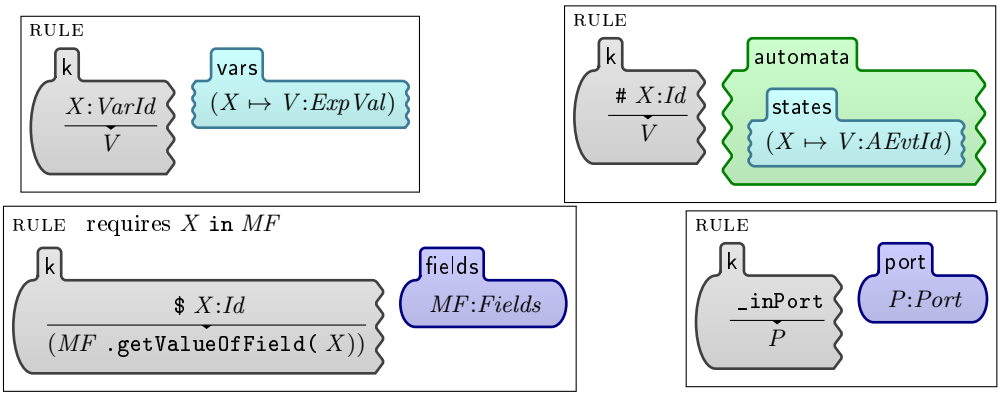


Figure 10: GPFL's semantics for variables

```
rule -- I:Int => ~Int I
rule I1:Int + I2:Int => I1 +Int I2
rule I1:Int - I2:Int => I1 -Int I2
rule I1:Int * I2:Int => I1 *Int I2
rule I1:Int / I2:Int => I1 /Int I2 requires I2 =/=Int 0

rule ! B:Bool => notBool B
rule B1:Bool & B2:Bool => B1 andBool B2
rule B1:Bool | B2:Bool => B1 orBool B2

rule I1:Int == I2:Int => I1 =Int I2
rule I1:Int < I2:Int => I1 <Int I2
rule I1:Int > I2:Int => I1 >Int I2
rule I1:Int <= I2:Int => I1 <=Int I2
rule I1:Int >= I2:Int => I1 >=Int I2

rule S1:String + S2:String => S1 +String S2
rule S1:String == S2:String => S1 ==String S2

rule P1:Port == P2:Port => P1 ==Port P2
```

Figure 11: $\mathbb{K}$ source file of GPFL expressions semantics

to the next thing to do, i.e. filter a packet or execute an interruption. Phase (c) executes the selected statement.

4.3.1 Program Initialization Phase.

As specified by the rules in Fig. 12, the execution of a GPFL program is initialized by splitting the program in two. The prologue goes into the `prg` cell and the filter statement goes into the `filter` cell. Then the prologue elements are executed one by one.

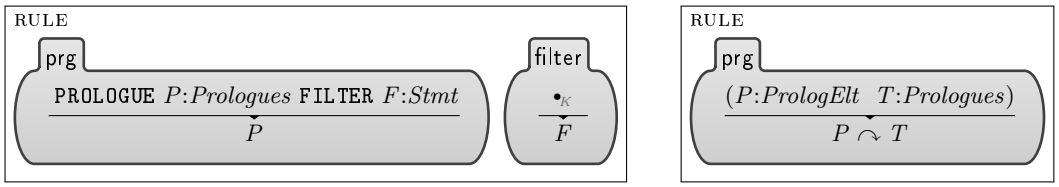


Figure 12: GPFL program top-level semantics in $\mathbb{K}$

The semantics of `AUTOMATON` prologues (Fig. 13) is to create a new `automataKindDef` cell containing the definition of the automata kind.

And, as specified by the rule in Fig. 14, the semantics of `INIT` prologues is to execute the associated statements. Any statement put in the `k` cell is to be executed, as specified in the remaining of this section.

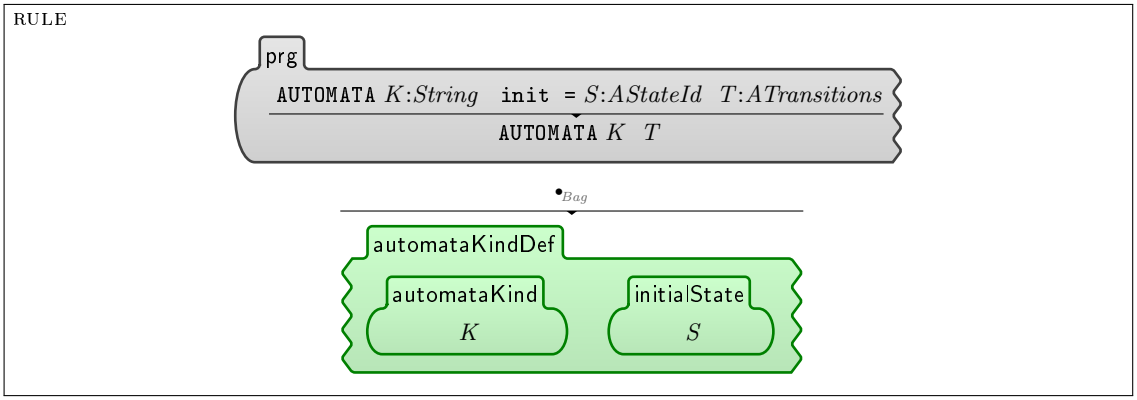
4.3.2 Statement Selection Phase.

Once the prologue as been executed, and after every execution of an interruption or after filtering a packet, the semantics of GPFL is to select the next statement to be executed: either the filtering statement (in the `filter` cell), or the statement associated with the next interruption if this interruption is triggered before the next packet arrives.

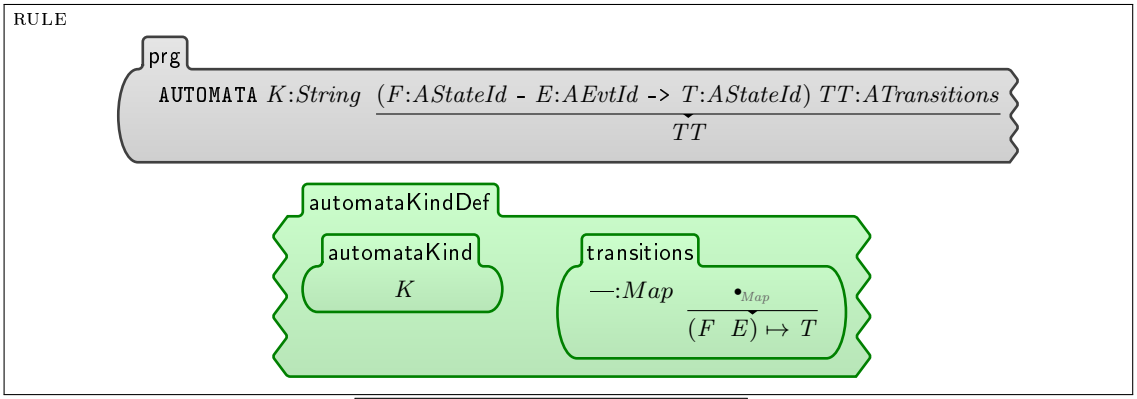
The process deciding which statement to execute next is specified in Fig. 15. This process starts by loading (from the `inTail` stream cell) and parsing the next packet to filter while resetting the `time`, `port` and `fields` cells. Then, depending on which event happens first between the arrival of a new packet and the triggering of the next interruption, one of two helper commands (`loadInterrupt` or `loadNextPkt`) is put into the `k` cell to indicate which action is to be taken next. It is to be noted that this process is not intrinsically part of the specification of GPFL's semantics. For practical reasons, this specification "pre-loads", into the `inHead` cell, future packets that have not arrived yet. An implementation of GPFL would not "pre-load" packets; for an implementation, the input stream (concatenation of the `inHead` and `inTail`) is a whole. This fact is reflected by the fact that all the rules of Fig. 15 are structural.

The formally specified semantics of `loadInterrupt` is shown in Fig. 16. The `loadInterrupt` command in the `k` cell is replaced by the statement associated with one of the interruptions scheduled to be triggered next (there can be many interruptions scheduled to be triggered at the same time). If the interruption triggered (I) is a recurring interruption, then I is scheduled to be triggered again at $T + P$ where T is the current time and P is the period of the recurring interruption I. Otherwise, the `interrupt` cell of I is simply removed.

Fig. 17 graphically displays the formal semantics of `loadNextPkt`. Every field of the packet P in the head of the input stream is loaded into the map of the `fields` environment cell. The `time` and `port` cells are set to the corresponding values associated to P. Finally, P is removed from the head of the input stream, and the filtering statement in the `filter` cell is loaded into the `k` cell in replacement of the `loadNextPkt` command.



SYNTAX *ATransitionPremise* ::= *AStateId* *AEvtId*



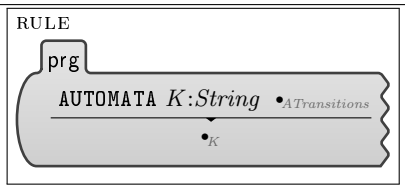


Figure 13: Automaton prologue semantics

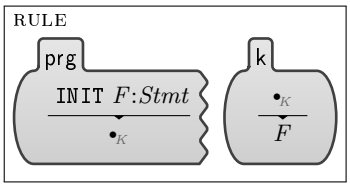


Figure 14: Initialization statement semantics

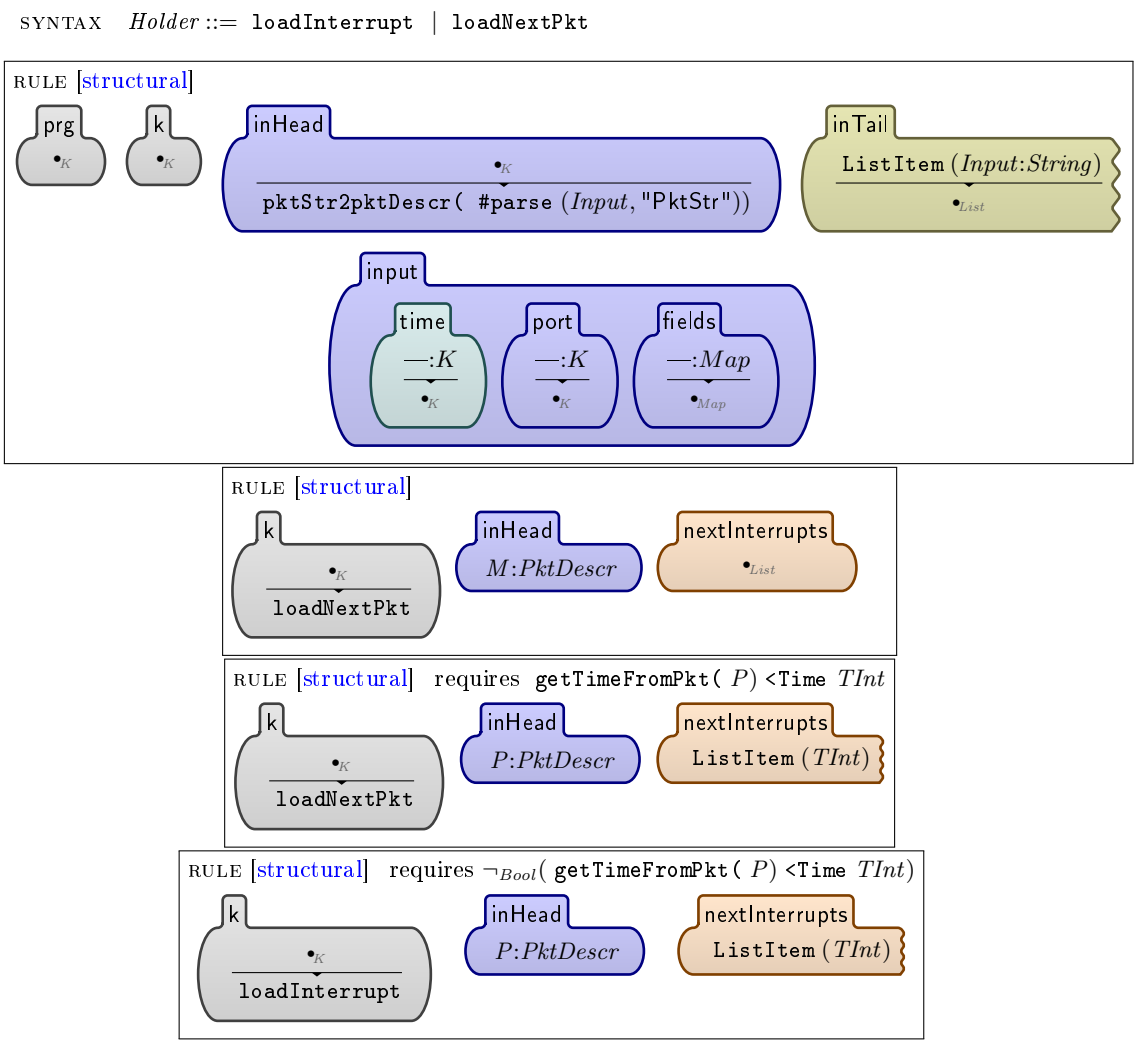


Figure 15: Semantic rules deciding which statement to execute next

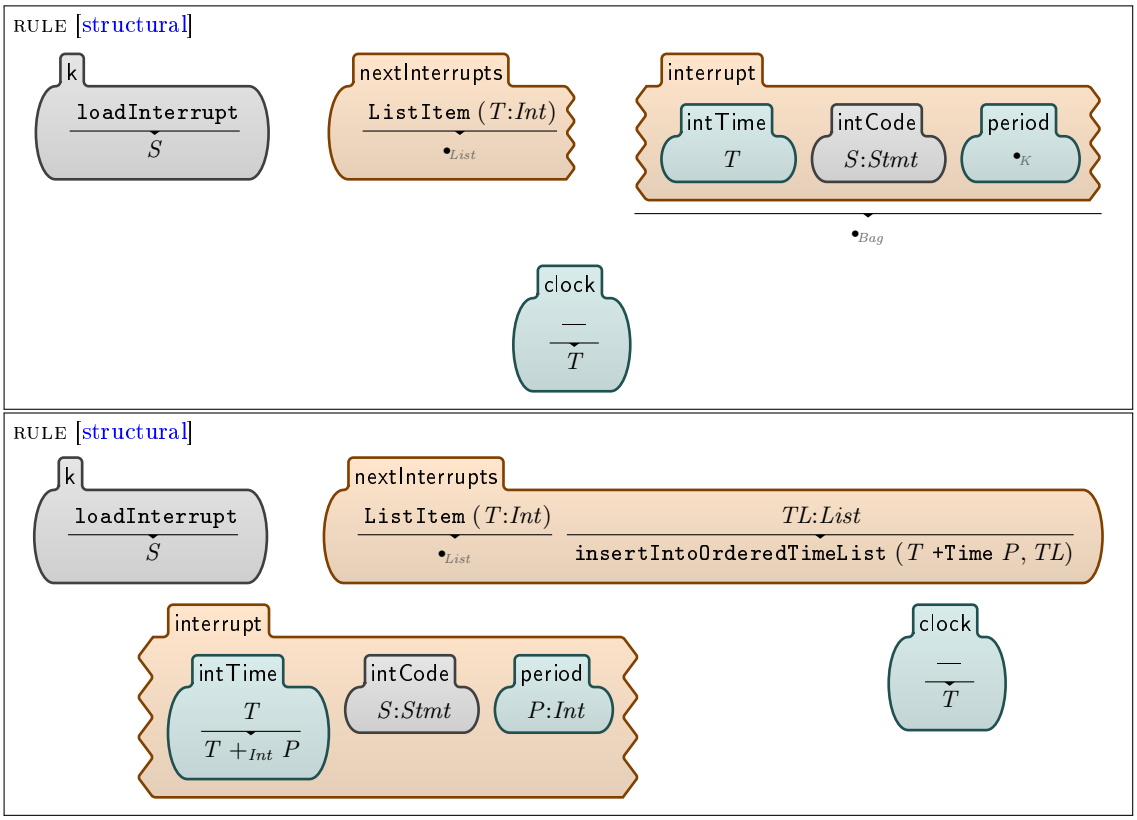


Figure 16: Semantics of `loadInterrupt`

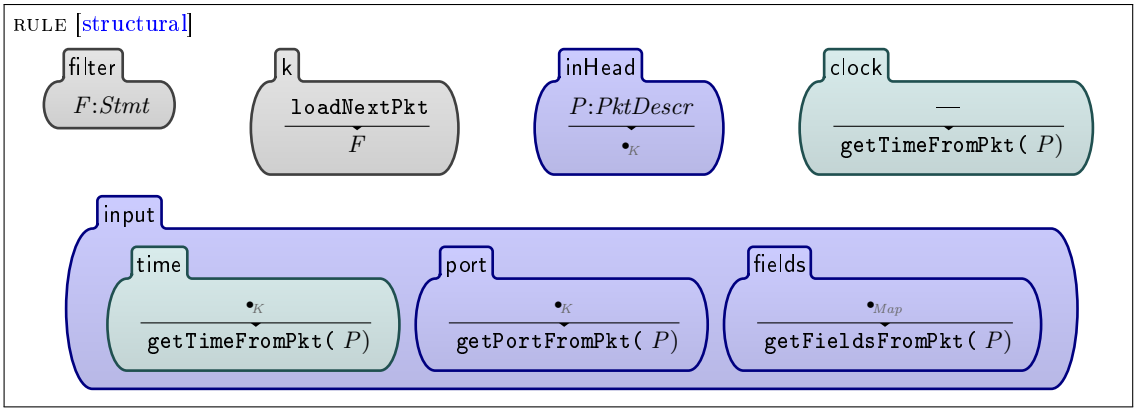


Figure 17: Semantics of `loadNextPkt`

4.3.3 Statement Execution Phase.

The definition of the `newInterrupt` statement semantics (Fig. 19) uses one helper function, named `insertIntoOrderedTimeList`, which inserts an integer into an ordered list (Fig. 18). As specified by the rules of Fig. 19, the semantics of a `newInterrupt` statement is simply to

SYNTAX $List ::=$ `insertIntoOrderedTimeList` $(Time, List)$ [function]

RULE
`insertIntoOrderedTimeList` $(I{:}Time, \bullet_{List})$
`ListItem` (I)

RULE requires I `<Time` J
`insertIntoOrderedTimeList` $(I{:}Time,$ `ListItem` (J) $T{:}List)$
`ListItem` (I) `ListItem` (J) T

RULE requires $\neg_{Bool}(I$ `<Time` $J)$
`insertIntoOrderedTimeList` $(I{:}Time,$ `ListItem` (J) $T{:}List)$
`ListItem` (J) `insertIntoOrderedTimeList` (I, T)

Figure 18: GPFL's `newInterrupt` helper semantics

create a new interruption in a new `interrupt` cell and to register this new interruption in the `nextInterrupts` cell.

GPFL's other statements semantics (specified in Fig. 20) is quite simple. To execute a pair of statements, the first statement is executed and then the second one. The `strict` attribute of the conditional statement (`cond`) syntax rule (Fig. 6) specifies that the guard of the statement must be evaluated to a value first; then the rules in Fig. 20 specify that the sub-statement is executed only if the guard is true. Similarly, the `strict` attribute of the iteration statement syntax rule specifies that the controlling expression must be evaluated to a value first. If this controlling value is 0 then the execution of the iteration statement is over; otherwise its sub-statement is executed once and the iteration statement is executed again with its controlling expression decreased by 1.

The semantics of the variable assignment command is quite standard (Fig. 21). The value associated to the variable in the map of the environment cell `vars` is updated to the new value of the variable. If the variable is not already present in the map of the `vars` cell, a structural rule adds it to the map, thus allowing the previous rule to apply.

The semantics of automata-related commands is given in Fig. 22. The `newAutomaton` command "creates" an new automaton of kind `K` and associates it to the variable `X`. The maps of the automata cell are updated to associate the kind `K` to the automaton referenced by `X`, and associate to `X` the initial state of automata of kind `K`. The `step` command sends the event `E` to the automaton referenced by `X`. If a transition triggered by `E` exists from the current state of the automaton, then the current state associated to `X` in the map of the `states` cell is updated with the new state; otherwise the error sub-statement `S` is executed.

The `alarm` command semantics is provided in Fig. 23. Its semantics is simply to generate a packet on the alarm output stream.

The packet related commands semantics (Fig. 24) relies on two internal commands: iSend, which sends a packet on the output stream; and iHalt, which halt the filtering process for the current packet. The `accept` command outputs the current packet and terminates the execution of the filter. The `drop` command terminates the execution of the filter. And `send` outputs a

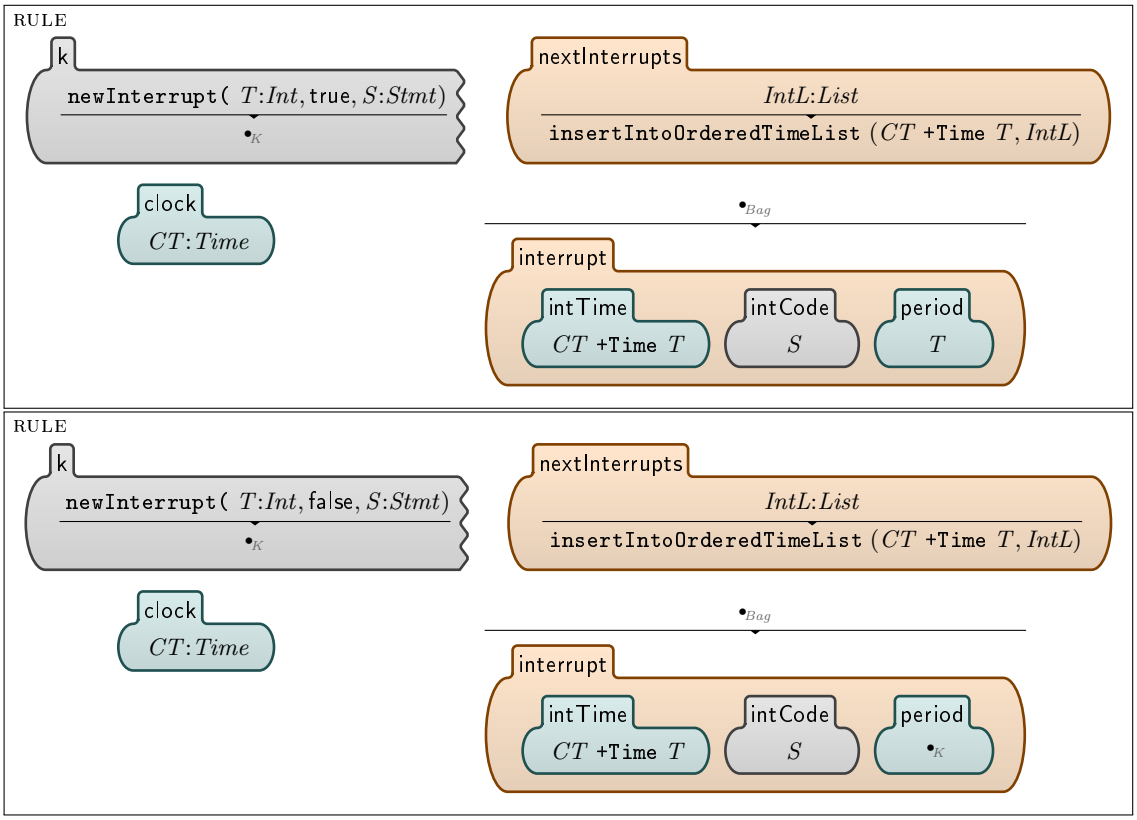


Figure 19: GPFL's `newInterrupt` statement semantics

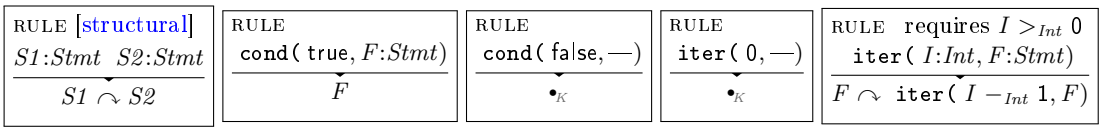


Figure 20: GPFL's other statements semantics

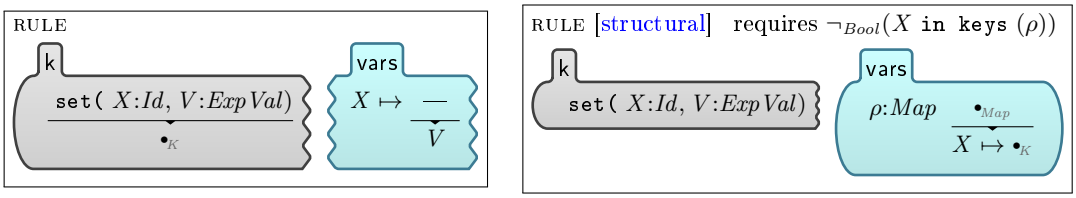


Figure 21: GPFL's `set` command semantics

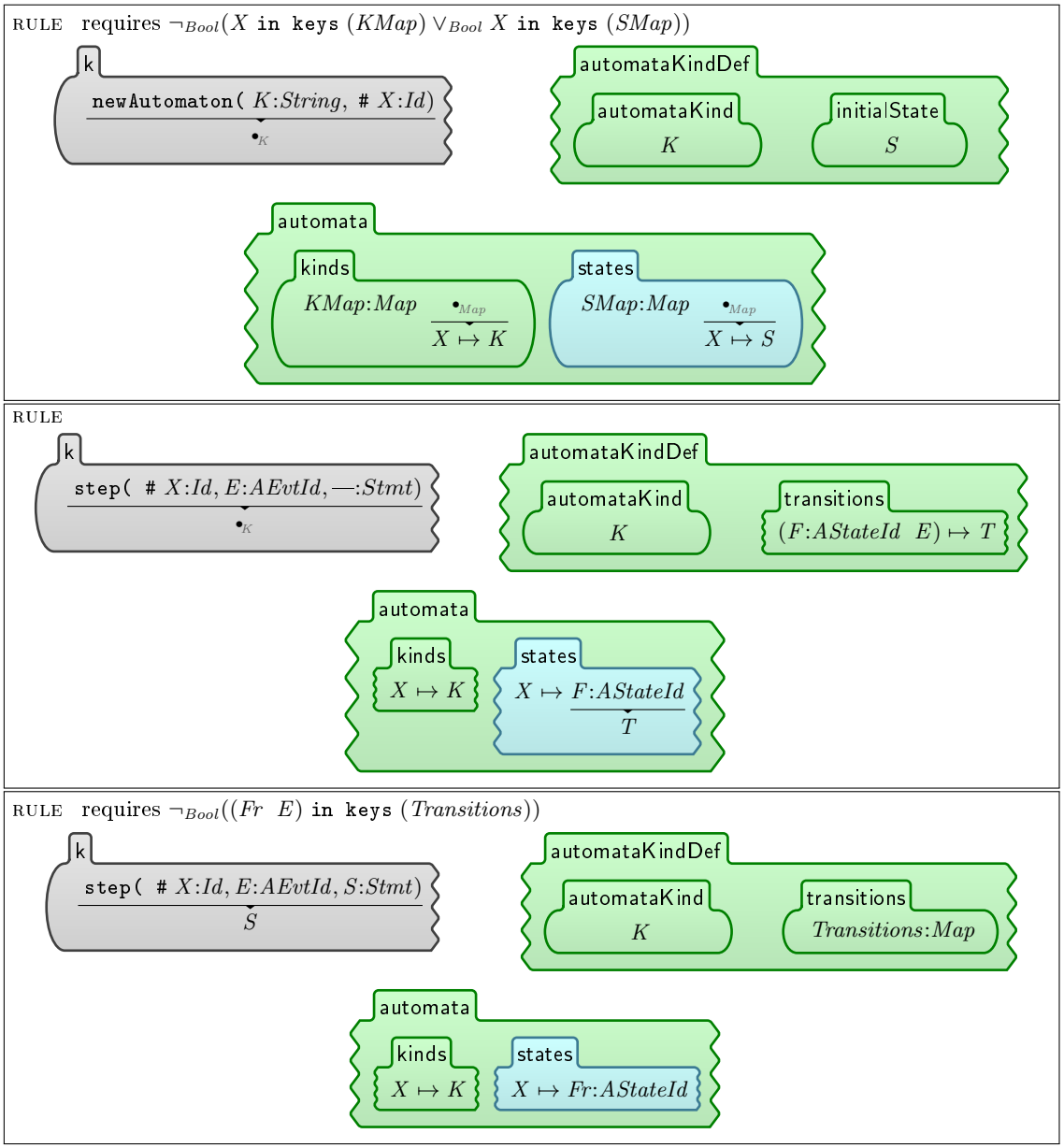


Figure 22: Automata commands semantics

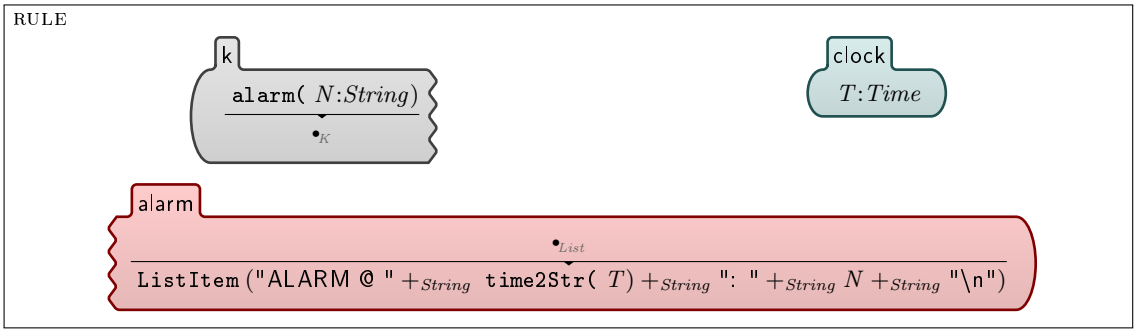


Figure 23: Alert commands semantics

packet.

SYNTAX *InternalCmd* ::= `iSend(` *Port*, *Fields*`)` | `iHalt`

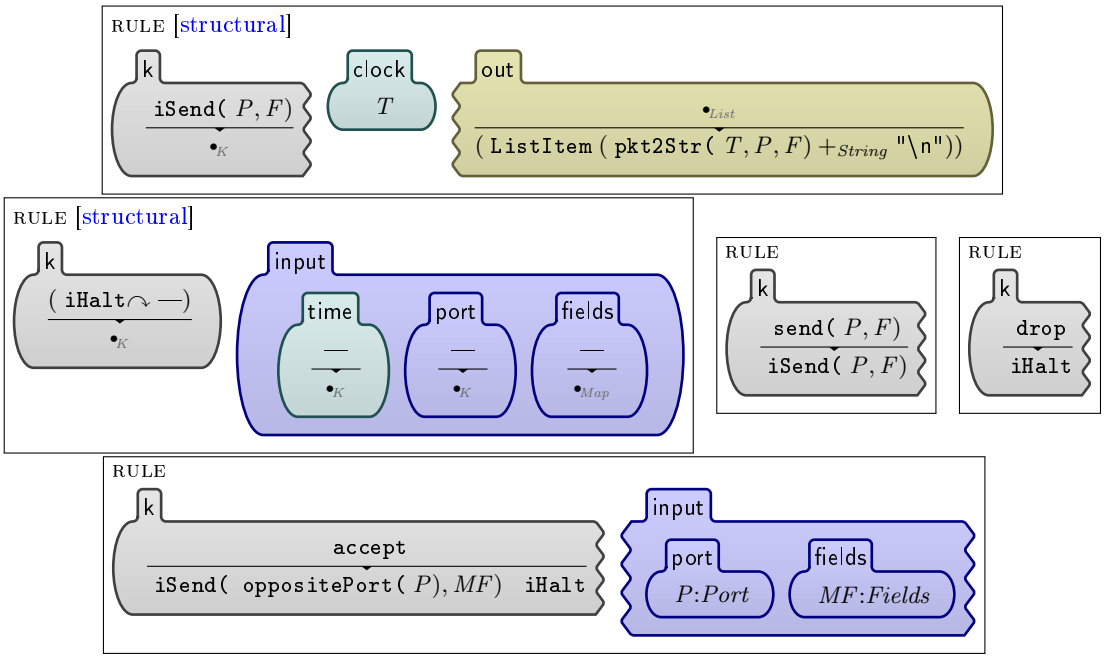


Figure 24: packet-related commands semantics

5 Testing GPFL's Specification

The above specification of GPFL syntax and semantics is not necessarily perfect. By a matter of fact, the imperfections of GPFL's specification are of interest to the experimentation reported in this paper. Indeed, the goal of the experimentation is to see how a tool such as the $\mathbb{K}$ framework can help to spot imperfections in filtering language specifications, and help correct them. One way to do so, is by "testing" the new language specified, which is possible if the framework used

to specify the language supports the execution or simulation of language specifications, which is the case for the $\mathbb{K}$ framework.

The test scenario used assumes a network of clients and servers. The clients request resources to servers using a made-up protocol called "DHCP cherry". The test scenario assumes that servers behave poorly when interacting concurrently with different clients. The objective of the test scenario is then to filter communications towards servers, as architectured in Fig. 25, in order to prevent any concurrent client-server interactions with any given server.

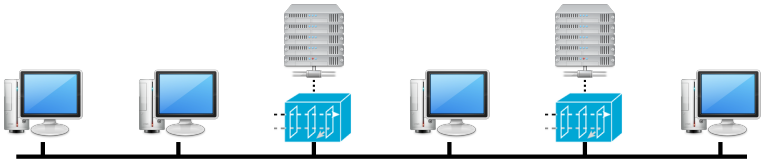


Figure 25: Network architecture of the test scenario

This test scenario is obviously made-up for this experimentation, which is a requirement due to confidentiality issues. However, it is still covering the most frequently used features of filtering languages similar to GPFL, while remaining simple enough for a sub-part of an experimentation.

5.1 DHCP cherry

The protocol used for this test scenario is a simplified version of the DHCP protocol. Packet formats and nominal sequences are described below.

5.1.1 Protocol

The protocol for DHCP cherry follows the packet sequences described in Fig. 26. The client

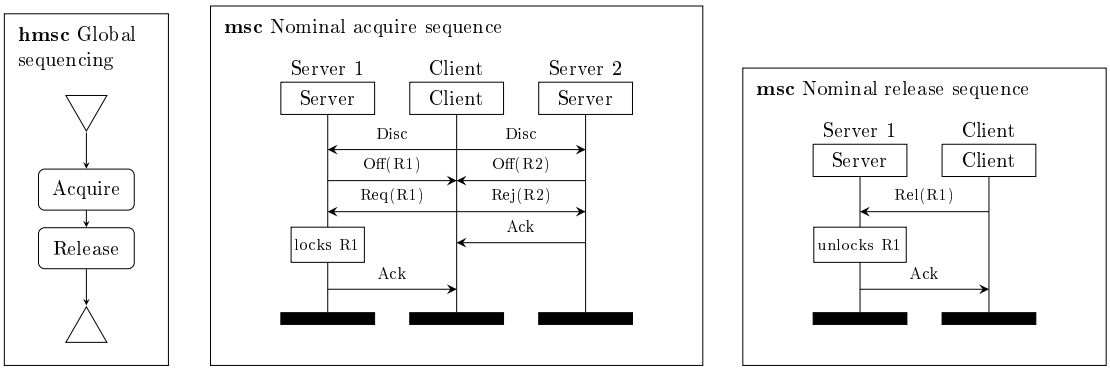


Figure 26: Nominal packet sequences of DHCP cherry protocol

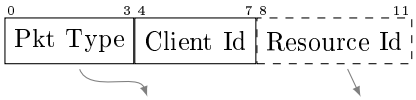
starts by broadcasting a request for resource (`Discover` packet). Servers answer with resource offers (Offer packet), but do not lock the resource for the client yet. The client chooses one of the offered resources ($R1$) and sends a request for that resource (Request packet) and rejections (Reject packet) for the other resources. Servers which received a rejection packet then send an acknowledgment packet (Acknowledge packet). The server, which received a request packet, locks the associated resource for the client and sends him an acknowledgment. The client is then free to use the resource for as long as he wishes. Once done with the resource, the client releases

the resource to the server (Release packet). And the server acknowledges reception of the release packet.

A client that does not receive any offer to a discovery request, or an expected acknowledgment, is supposed to try again later to emit the packet to which it did not receive an answer. However, there is no explicit recovery mechanism in the protocol. If a packet sequence stops between the Request packet and the Release packet, the associated resource is "lost".

5.1.2 Packet formats

The format of packets is given in Fig. 27. A packet is 8 or 12 bits long. A packet starts by a 4



Packet	MSC	Type encoding	Resource part	Pkt length
Discover	Disc	0	NO	8 bits
Offer	Off(R)	1	YES	12 bits
Request	Req(R)	2	YES	12 bits
Reject	Rej(R)	3	YES	12 bits
Release	Rel(R)	4	YES	12 bits
Acknowledge	Ack	5	NO	8 bits

Figure 27: Format of DHCP cherry packets

bits long packet type identifier (Discover, Offer, ...), followed by a 4 bits long client identifier identifying the client involved in the session. If the packet carries a resource identifier, a 4 bits long resource identifier is appended at the end of the packet.

5.2 The Filtering Policy to Enforce

From the point of view of servers, non-concurrent interactions are sequential instances of only three generic atomic packet sequences. Those atomic packet sequences are the ones accepted by the automaton in Fig. 28. In this automaton, "in:MP", resp. "out:MP", is a transition trigger

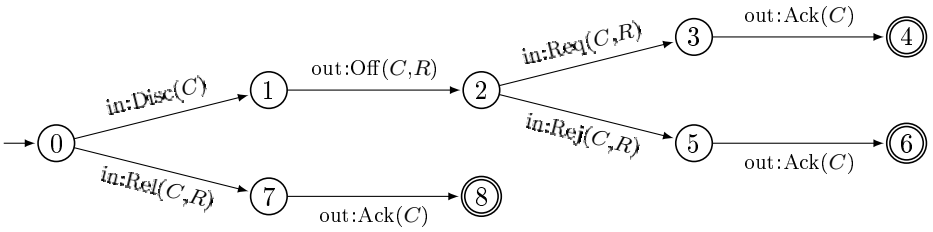


Figure 28: Automaton of Server-side Atomic Packet Sequences

matching any incoming packet (from the rest of the network to the server), resp. outgoing packet (from the server to the rest of the network), matching packet pattern MP. C and R are variables. C is a client identifier variable. R is a resource identifier variable. C, resp. R, has to be instantiated in the same way (have the same value) for any packet of the same atomic packet sequence accepted by the automaton.

The automaton of Fig. 28 is refined into a filtering policy automaton described in Fig. 29. Variables C and R have the same constraints as for the automaton of Fig. 28. The variable "$*$" matches any value, packet pattern "out:$*$" matches any outgoing packet, and packet pattern "out:$*$ - Ack(C)" matches any outgoing packet except Ack(C). This filtering policy accepts every

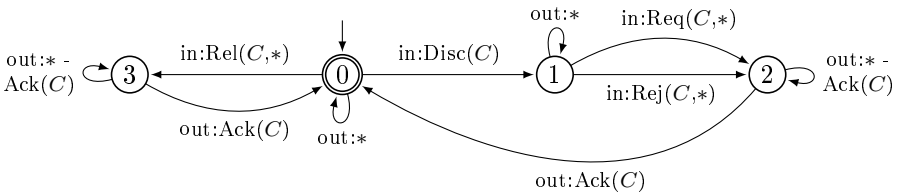


Figure 29: Filtering Policy Automaton

outgoing packet; thus having no effect on the packets generated by the server. For incoming packets, if the current state of the automaton has no transition whose trigger matches the packet then the packet is discarded; otherwise, the packet is accepted and the associated transition is triggered.

This filtering policy assumes that clients comply with the DHCP cherry protocol and ensures only that the filtered server only interacts sequentially with clients. If there is no idle server ready to receive a packet from a client, this client gets no answer and is expected to retry later.

5.3 The Filter Code in GPFL

The GPFL's code for this filtering policy is contained the file `dhcp.gpfpl`, displayed below.

```
PROLOGUE
  AUTOMATA "DHCP incoming controller"
    init = "0"
    "0" -"Disc"-> "1"
    "1" -"Req"-> "2"
    "1" -"Rej"-> "2"
    "2" -"Ack"-> "0"
    "0" -"Rel"-> "3"
    "3" -"Ack"-> "0"
  INIT
    newAutomaton("DHCP incoming controller", #A)
    set(ignoredPktCnt, 0) set(ignoredPktThreshold, 5)
    newInterrupt(60, true, set(ignoredPktCnt, 0))

FILTER
  cond(_inPort == inSide,
    cond( ($pktType == "Ack") & ($clientId == currentClient),
      step(#A, "Ack", nop)
      set(currentClient, "")
    )
    accept
  )
  cond(_inPort == outSide,
    cond($pktType == "Disc",
      step(#A, "Disc",
        set(ignoredPktCnt, ignoredPktCnt + 1)
        cond(ignoredPktCnt >= ignoredPktThreshold,
          alarm("Many external messages ignored!")
          set(ignoredPktCnt, 0)
```

```
30          )
            drop
32        )
          set(currentClient, $clientId)
34        accept
      )
36    cond( ($pktType == "Req") | ($pktType == "Rej"),
        cond(! ($clientId == currentClient),
38        set(ignoredPktCnt, ignoredPktCnt + 1)
          cond(ignoredPktCnt >= ignoredPktThreshold,
40          alarm("Many external messages ignored!")
            set(ignoredPktCnt, 0)
42        )
          drop
44      )
        step(#A, $pktType,
46        set(ignoredPktCnt, ignoredPktCnt + 1)
          cond(ignoredPktCnt >= ignoredPktThreshold,
48          alarm("Many external messages ignored!")
            set(ignoredPktCnt, 0)
50        )
          drop
52      )
        accept
54    )
      cond($pktType == "Rel",
56      step(#A, "Rel",
          set(ignoredPktCnt, ignoredPktCnt + 1)
58        cond(ignoredPktCnt >= ignoredPktThreshold,
            alarm("Many external messages ignored!")
60          set(ignoredPktCnt, 0)
          )
62        drop
        )
64      set(currentClient, $clientId)
        accept
66    )
      drop
68  )
    alarm("Unhandled message") drop
```

The states of the filtering automaton of Fig. 29 are directly encoded in an automata kind definition in the prologue, with generic triggering conditions that only encode the type of packet received. A unique instance (`#A`) of this kind of automata is created. For every packet received by the filter, additional triggering conditions (packet input port, client identifier, ...) are handled in the `FILTER` code itself. If a packet is received with a type compatible with additional triggering conditions, the packet type is sent to the automaton `#A` to verify that the current state is compatible with the reception of this type of packet, and update the state of the automaton.

In addition, every dropped packet increments a counter (`ignoredPktCnt`), which is reset to 0 every 60 "time unit" by a recurrent interruption initialized in the prologue. If this counter reaches the threshold (5), an alarm is raised to warn that "many" packets are dropped by the filter.

5.4 Simulating the Filter

The above filtering code written in GPFL can then be simulated by running the following command (in Linux Bash) :

```
krun dhcp.gpfpl < dhcp_input-dataset.txt > dhcp_output.txt
```

where `dhcp_input-dataset.txt` contains a sequence of "parsed" packets (decoded packets, Fig. 5) input to the filter. The output of the simulation of the code (`dhcp.gpfpl`) written in the specified language (GPFL) is written in `dhcp_output.txt`.

The following input (`dhcp_input-dataset.txt`):

```
(002;outSide;pktType="Disc",clientId="A")
(005;inSide;pktType="Off",clientId="A",ressourceId="D")
(006;outSide;pktType="Disc",clientId="7")
(009;outSide;pktType="Off",clientId="7",ressourceId="5")
(012;outSide;pktType="Req",clientId="7",ressourceId="5")
(014;outSide;pktType="Disc",clientId="F")
(015;outSide;pktType="Req",clientId="A",ressourceId="D")
(017;outSide;pktType="Ack",clientId="7")
(018;inSide;pktType="Ack",clientId="A")
(102;outSide;pktType="Disc",clientId="B")
(105;inSide;pktType="Off",clientId="B",ressourceId="E")
(106;outSide;pktType="Disc",clientId="7")
(109;outSide;pktType="Off",clientId="7",ressourceId="5")
(112;outSide;pktType="Req",clientId="7",ressourceId="5")
(114;outSide;pktType="Disc",clientId="F")
(115;outSide;pktType="Rej",clientId="B",ressourceId="E")
(117;outSide;pktType="Ack",clientId="7")
(124;outSide;pktType="Disc",clientId="F")
(138;inSide;pktType="Ack",clientId="B")
(202;outSide;pktType="Rel",clientId="A",ressourceId="D")
(205;outSide;pktType="Disc",clientId="F")
(206;outSide;pktType="Disc",clientId="7")
(207;outSide;pktType="Disc",clientId="F")
(209;outSide;pktType="Off",clientId="7",ressourceId="5")
(211;outSide;pktType="Disc",clientId="F")
(212;outSide;pktType="Rej",clientId="7",ressourceId="5")
(214;outSide;pktType="Disc",clientId="F")
(216;outSide;pktType="Ack",clientId="7")
(217;outSide;pktType="Disc",clientId="F")
(218;inSide;pktType="Ack",clientId="A")
(324;outSide;pktType="Disc",clientId="F")
```

produces the following expected output (`dhcp_output.txt`):

```
(2; inSide; pktType="Disc",clientId="A")
(5; outSide; ressourceId="D",pktType="Off",clientId="A")
(15; inSide; ressourceId="D",pktType="Req",clientId="A")
(18; outSide; pktType="Ack",clientId="A")
(102; inSide; pktType="Disc",clientId="B")
(105; outSide; ressourceId="E",pktType="Off",clientId="B")
(115; inSide; ressourceId="E",pktType="Rej",clientId="B")
(138; outSide; pktType="Ack",clientId="B")
(202; inSide; ressourceId="D",pktType="Rel",clientId="A")
ALARM @ 212: Many external messages ignored!
(218; outSide; pktType="Ack",clientId="A")
(324; inSide; pktType="Disc",clientId="F")
```

appended with a description of the final configuration (which is not displayed here).

6 Discussion on the Experimentation

The primary goal of this paper is not to set out the filtering policy described in Sect. 5 or, even, GPFL's specification described in Sect. 4. This paper is an experience report on a primary

evaluation of the cost and benefits of using formal specification tools in general, and the $\mathbb{K}$ framework in particular, to formally specify the syntax and semantics of filtering languages. Overall, it seems to the authors that using the $\mathbb{K}$ framework helped greatly to improve GPFL's specification quality. It forced the specification authors to be precise, and helped spot various errors and missing specification fragments.

With regard to the "cost", this experimentation argues in favor of tool supported formal specifications for high quality specifications of filtering languages. Of course, using natural language, it is possible to produce a cheaper, but ambiguous and approximate, specification. However, it is the opinion of the authors that using natural language to produce a specification with a similar level of precision and correctness would be more costly. With a decent knowledge of operational semantics concepts, the cost for newcomers to the $\mathbb{K}$ framework is relatively low, thanks to the numerous tutorials (in text and video), manuals and examples.

Compared to formal specification without tool support, the cost of the constraints imposed by the $\mathbb{K}$ framework seems to the authors to be lower than the "benefits" provided by the tool support. Typically, the ability to "execute" the formal specification of the filtering language requires a particular handling of input/output related rules. However, this same ability to "execute" the formal specification of the filtering language is highly beneficial when validating the correctness of the specification and expressivity of the language.

Other benefits of tool supported formal specifications of languages are numerous. In natural language documents specifying new languages, it is too common for program examples to be inconsistent with the language grammar. It is easily explained by the modifications brought to the language grammar during the specification document development. Examples directly related to the modified statements are usually modified accordingly. However, examples related to other aspects of the language are often forgotten. Using a tool supported formal specification, it is easy to adopt a "continuous/frequent integration" approach where examples are: written in separate files, regularly parsed to verify that they comply with the current grammar, and automatically imported in the specification document (the creation of this paper used this approach).

Additionally, use of a tool-supported formal specification approach modifies the workflow often applied when using natural language specification documents. With natural language specifications, the specification document writing process usually starts early after a short engineering phase (it may not be true for a language *development* process, however it is often the case in "pure" language *specification* processes), and the main part of the language specification is done during the specification document writing process. With a tool-supported formal specification approach, the specification of the language tend to be first developed inside the tool, and then the language specification is clarified during the specification document writing process. With a tool-supported formal specification approach, the language specification becomes a two phases process with two different views on the language specification. The "two different views" aspect is particularly true with the $\mathbb{K}$ framework were semantics rules are entered textually in the source file and can be rendered graphically for the specification document. This two phases workflow (development then clarification and documentation) helps spot: differences of treatments (in particular for configuration cells), generalization and reuse opportunities (for example, in this experimentation, the use of only two internal commands, `iSend` and `iHalt`, to encode the three packet commands `accept`, `drop` and `send`), different concepts that are candidates to modularization (for example, in this experimentation, the externalization of packet data type definitions and string conversions), errors that manifest themselves in rare occasions (for example, in an earlier version of GPFL, automaton states and variable values where stored in the same map, which could trigger a key clash caused by variable and automaton identifiers having the same "name" part), or general simplifications (for example, during this report writing process, GPFL's configuration has been heavily reformatted to simplify the language specification and be closer

to the concepts manipulated). From the authors experience, in general, a tool-supported formal specification helps simplify and clarify a language specification.

Moreover, the ability to execute the formal specification allows to adopt an incremental approach for the specification of the different statements semantics. In such an approach, the syntax of the language is first specified. Then a program example making use of all the statements of the language in as much context as reasonable is written. The semantics of the statements is then defined statements by statements. The program is executed using $\mathbb{K}$'s run time; and the execution stops when reaching a statement whose semantics is not defined yet. All the semantics rules associated to this statement are then defined. When stopping an execution, $\mathbb{K}$'s run time displays the current state of the configuration which can help specify the missing semantics rules. As the test program execution goes further and further during the language semantics specification process, this incremental approach is more rewarding for people in charge of the specification. The impact of using this incremental approach (which is not required by the $\mathbb{K}$ framework) on the quality of the specifications produced remain to be investigated.

Finally, the ability to execute the formal specification allows to test and validate the language specification. Two important points to validate are: the expressivity of the language and its expected semantics. The GPFL's code provided in Sect. 5.3 emphasizes the limitations of the simple automata that can be defined using GPFL. It could be useful to have automaton state variables, and triggering conditions that test and check automaton state variable values. However, adding automaton state variables would complexify automata definitions. Similarly, the GPFL code provided in Sect. 5.3 contains a recurring code sequence to handle alarms that are triggered only a threshold of a specific event occurrences is reached. It could be useful to add a specific command to GPFL which would have the same semantics as this recurring sequence. The ability to test programs does not solve expressivity questions (which have to be answered on a per language basis), however it helps explicit those questions. With regard to expected semantics, writing test programs helps validate that programs have the semantics that users would expect. The initial version of the filter code provided in Sect. 5.3 did not behave as expected. It ended up being a misplaced statement in the filter code, but could also have been a problem with the semantics specification. Discovering the cause of a misbehavior of a test program (error in the semantics or the program) could be greatly simplified by $\mathbb{K}$'s debugger which can "execute" formal specifications step by step; especially as Domain Specific Languages (specifications and implementations) usually have limited debugging facilities (which is in accordance with their philosophy of limited expressivity for the sake of simplification). However, sadly, $\mathbb{K}$'s debugger crashed on our program with the version of the $\mathbb{K}$ framework used for this experimentation (version 3.6). This can be explained by the fact that $\mathbb{K}$ development effort is now focused on the next version to come (version 4.0). The authors plan to migrate this experimentation once version 4.0 exits the beta stage. Finally, the ability to execute the formal specification helps to validate a set of test programs that can be used as smoke test for language implementations.

7 Conclusion

This paper reports on an experiment to formally specify the syntax and semantics of a filtering language (GPFL) using the tool-supported framework $\mathbb{K}$. The filtering language specified in this report has been made up for this experimentation; however, it covers the majority of concepts usually encountered in filtering languages. No comparison between different tools is made in this experiment. The goal of the experiment is to study the feasibility of using a tool-supported formal approach for the specification of domain-specific filtering languages having a complexity similar to filtering languages encountered in real-life projects.

The $\mathbb{K}$ framework proved to be sufficiently expressive to naturally express the syntax and semantics of GPFL in a formal way. The effort required by this formal specification is judged reasonable by the authors, and within reach of average engineers which have been exposed previously to operational semantics theories. Newcomers life is made easier by the numerous manuals, examples and tutorials available for the $\mathbb{K}$ framework. The tool support is a welcome help during the specification process. In particular, the ability to execute (or simulate) $\mathbb{K}$ formal specifications helps greatly when developing and fine tuning the language specification, and when producing smoke tests for the implementation.

Following such a specification process may seem to be in complete contradiction to any agile development principles [4]. However, using a tool-supported *executable* specification methodology allows to comply with one of the pillars of agile development: *early feedback*. As the language specification is executable, it is possible to ask final users (if some are available) to test the language and provide feedbacks on different aspects of the language, including its expressivity. In fact, IBM's Continuous Engineering development methodology [22] advocates for the use of executable models at every steps of the development.

With regard to the benefits of putting the effort to produce a *formal* specification, the authors opinion, on improved quality and usefulness of formal specifications compared to non formal specifications written in natural language, is relatively well summarized in the following statement by David Schmidt [20].

> "Since data structures like symbol tables and storage vectors are explicit, a language's subtleties are stated clearly and its flaws are exposed as awkward codings in the semantics. This helps a designer tune the language's definition and write a better language manual. With a semantics definition in hand, a compiler writer can produce a correct implementation of the language; similarly, a user can study the semantics definition instead of writing random test programs."
>
> David Schmidt in ACM Computing Surveys [20]

This statement is supported by the numerous ambiguities in common programming languages, like C/C++ or Java. Some of those ambiguities, as the memory model of multi-threaded Java™ programs [11], required a formal specification in order to be solved.

> "Unfortunately, the current specification has been found to be hard to understand and has subtle, often unintended, implications. Certain synchronization idioms sometimes recommended in books and articles are invalid according to the existing specification. Subtle, unintended implications of the existing specification prohibit common compiler optimizations done by many existing Java virtual machine implementations. [...] Several important issues, [...] simply aren't discussed in the existing specification."
>
> JSR-133 expert group [11]

In the experimentation reported in this paper, no formal analysis of the formal specification produced has been attempted. In future work, the authors plan to try some of the experimental tools available with the $\mathbb{K}$ framework on GPFL's specification. If time allows, a similar experimentation could be repeated with other tools oriented toward the formal specification of languages.

References

[1] Backus, J.W.: The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. In: Proc. Int. Conf. Information Processing. pp. 125–132. UNESCO (1959)

[2] Basten, H.J.S., van den Bos, J., Hills, M.A., Klint, P., Lankamp, A.W., Lisser, B., van der Ploeg, A.J., van der Storm, T., Vinju, J.J.: Modular Language Implementation in Rascal – Experience Report. Science of Computer Programming 114, 7–19 (Dec 2015)

[3] Chalub, F., Braga, C.: Maude MSOS Tool. In: Proc. Int. Work. Rewriting Logic and its Applications. Electronic Notes in Theoretical Computer Science, vol. 176, pp. 133–146. Elsevier Science Publishers B. V. (2007)

[4] Cockburn, A.: Agile Software Development: The Cooperative Game. Pearson Education, 2nd edn. (2007)

[5] Dubuisson, O.: ASN.1 – Communication between Heterogeneous Systems. OSS Nokalva (Jun 5, 2000), `http://www.oss.com/asn1/dubuisson.html`, translated from French by Philippe Fouquart

[6] Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press (Jul 2009)

[7] van Heijenoort, J.: From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931, chap. Peano (1889). The principles of arithmetic, presented by a new method. Source Books in the History of the Sciences, Harvard University Press (Jan 2002), a translation and excerpt of Peano's 1889 paper "Arithmetices principia, nova methodo exposita"

[8] Jézéquel, J.M., Barais, O., Fleurey, F.: Summer School on Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 6491, chap. Model Driven Language Engineering with Kermeta, pp. 201–221. Springer Berlin Heidelberg (2011)

[9] Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of metalanguages and its implementation in the Kermeta language workbench. Software & Systems Modeling 14(2), 905–920 (2013)

[10] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 6, Telecommunications and information exchange between systems: Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. International Standard 8824-1, ISO/IEC (Nov 2015), ISO/IEC version of ITU-T X.680 (08/2015)

[11] JSR-133 expert group: JSR-133 Java™ Memory Model and Thread Specification Revision. Java Specification Request (JSR) 133, Sun Microsystems, Inc. (30 Sep 2004), `https://jcp.org/en/jsr/detail?id=133`

[12] Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: Run Your Research: On the Effectiveness of Lightweight Mechanization. In: Proc. Symp. Principles of Programming Languages. SIGPLAN Not., vol. 47, pp. 285–296. ACM, New York, NY, USA (Jan 2012)

[13] Klint, P.: Tribute to a great Meta-Technologist: from Centaur to The Meta-Environment. In: Bertot, Y., Huet, G., Levy, J.J., Plotkin, G. (eds.) From Semantics to Computer Science, Essays in Honour of Gilles Kahn, pp. 235–264. Cambridge University Press (2009)

[14] Klint, P., Vinju, J.J., Hills, M.A.: RLSRunner: Linking Rascal with K for Program Analysis. In: Proc. Int. Conf. Software Language Engineering. Springer (2011)

[15] Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proc. Int. Working Conf. Source Code Analysis and Manipulation. pp. 168–177. IEEE Computer Society (2009)

[16] Knuth, D.E.: Backus Normal Form vs. Backus Naur Form. Commun. ACM 7(12), 735–736 (Dec 1964)

[17] Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable Engineering of Real-world Semantics. In: Proc. Int. Conf. Functional Programming. SIGPLAN Not., vol. 49, pp. 175–188. ACM (2014)

[18] Roşu, G., Şerbănuţă, T.F.: An Overview of the $\mathbb{K}$ Semantic Framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)

[19] Roşu, G., Şerbănuţă, T.F.: $\mathbb{K}$ Overview and SIMPLE Case Study. In: Proc. Int. Work. K Framework and its Applications (K 2011). Electronic Notes in Theoretical Computer Science, vol. 304, pp. 3–56 (2014)

[20] Schmidt, D.A.: Programming Language Semantics. ACM Computing Surveys 28(1) (Mar 1996)

[21] Sewell, P., Nardelli, F.Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strniša, R.: Ott: Effective Tool Support for the Working Semanticist. J. Functional Programming 20(1), 71–122 (Jan 2010)

[22] Shamieh, C.: Continuous Engineering For Dummies®. IBM Limited Edition, John Wiley & Sons, Inc (2014)

[23] van der Storm, T., Vinju, J.J.: Using the Meta-Environment for Domain Specific Language Engineering. Tech. Rep. SEN-R0805, CWI Software Engineering (2008)

[24] Şerbănuţă, T.F., Arusoaie, A., Lazar, D., Ellison, C., Lucanu, D., Roşu, G.: The $\mathbb{K}$ Primer (version 3.3). Electronic Notes in Theoretical Computer Science 304, 57–80 (2014), proc. Int. Work. K Framework and its Applications (K 2011)

A Full $\mathbb{K}$ code

The following three listings contain the full $\mathbb{K}$ code specifying GPFL.

```
module GPFPL-DATA

  syntax Time ::= Int

  syntax Time ::= Time "+Time" Time [function]
  rule T1:Int +Time T2:Int => T1 +Int T2 [structural]

  syntax Bool ::= Time "<Time" Time [function]
  rule T1:Int <Time T2:Int => T1 <Int T2 [structural]

  syntax Port ::= "inSide" | "outSide"

  syntax Bool ::= Port "==Port" Port [function]
  rule P1:Port ==Port P2:Port => P1 ==K P2 [structural]

  syntax Port ::= "oppositePort(" Port ")" [function]
  rule oppositePort( inSide:Port ) => outSide
  rule oppositePort( outSide:Port ) => inSide

  syntax Fields ::= Map

  syntax Bool ::= Id "in" Fields [function]
  rule X:Id in MF:Map => (X in keys(MF)) [structural]

  syntax K ::= Fields ".getValueOfField(" Id ")" [function]
  rule MF:Map .getValueOfField( X:Id )  => MF[X] [structural]

  syntax PktDescr ::= "(" Time "," Port "," Fields ")"

  syntax Time ::= "getTimeFromPkt(" PktDescr ")" [function]
  rule getTimeFromPkt( ( T:Time , _:Port , _:Fields ) ) => T

  syntax Port ::= "getPortFromPkt(" PktDescr ")" [function]
  rule getPortFromPkt( ( _:Time , P:Port , _:Fields ) ) => P

  syntax Fields ::= "getFieldsFromPkt(" PktDescr ")" [function]
  rule getFieldsFromPkt( ( _:Time , _:Port , MF:Fields ) ) => MF

endmodule
```

Listing: `dataDefs.k3`

```
require "dataDefs.k3"

module STRING-CONVERSIONS

  imports GPFPL-DATA

  syntax TimeStr ::= Int

  syntax String ::= "time2Str(" Time ")" [function]
  rule time2Str( T:Int ) => Int2String( T )
  syntax Time ::= "str2Time(" TimeStr ")" [function]
  rule str2Time( T:Int ) => T

  syntax PortStr ::= Port

  syntax String ::= "port2Str(" Port ")" [function]
  rule port2Str( inSide ) => "inSide"
  rule port2Str( outSide ) => "outSide"
  syntax Port ::= "str2Port(" PortStr ")" [function]
  rule str2Port( P:Port ) => P

  syntax FieldStr ::= Id "=" String
  syntax FieldsStr ::= List{ FieldStr , "," }

  syntax String ::= "fields2Str(" Fields ")" [function]
  rule fields2Str( .Map ) => ""
```

```
  rule fields2Str( F:Id |-> V:String ) => ( Id2String(F) +String "=" +String "\"" +
     String V +String "\"" )
  rule fields2Str( F:Id |-> V:String FTail:Map ) => ( fields2Str(F |-> V) +String "," +
     String fields2Str(FTail) )

  syntax Fields ::= "str2Fields(" FieldsStr ")" [function]
  rule str2Fields( M:FieldsStr ) => str2mfInternals(M)
  syntax Map ::= "str2mfInternals(" FieldsStr ")" [function]
  rule str2mfInternals( .:FieldsStr ) => .Map
  rule str2mfInternals( F:Id = V:String ) => (F |-> V)
  rule str2mfInternals( F:Id = V:String , MFS:FieldsStr ) => (F |-> V) str2mfInternals(
     MFS)

  syntax PktStr ::= "(" TimeStr ";" PortStr ";" FieldsStr ")"

  syntax String ::= "pkt2Str(" Time "," Port "," Fields ")" [function]
  rule pkt2Str( T:Time , P:Port , M:Map ) => ( "(" +String time2Str(T) +String "; " +
     String port2Str(P) +String "; " +String fields2Str(M) +String ")" )

  syntax PktDescr ::= "pktStr2pktDescr(" PktStr ")" [function]
  rule pktStr2pktDescr( ( T:TimeStr ; P:PortStr ; MF:FieldsStr ) ) => ( str2Time(T) ,
     str2Port(P) , str2Fields(MF) )

endmodule
```

Listing: `stringConversions.k3`

```
require "dataDefs.k3"
require "stringConversions.k3"

module GPFPL-SYNTAX

  imports GPFPL-DATA
  imports STRING-CONVERSIONS

  syntax ExpVal ::= Int | Bool | String | AEvtId | Port

  syntax BuiltInId ::= "_inPort"
  syntax VarId ::= Id
  syntax FieldId ::= "$" Id
  syntax AutomatonId ::= "#" Id
  syntax ExpId ::= BuiltInId | VarId | FieldId | AutomatonId

  syntax UnaryOp ::= "--" | "!"
  syntax BinaryOp ::= "+" | "-" | "*" | "/" | "&" | "|"
                    | "==" | "<" | ">" | "<=" | ">="

  syntax Exp ::= ExpVal | ExpId
               | UnaryOp Exp            [strict(2)]
               | Exp BinaryOp Exp       [strict(1,3), left]
               | "(" Exp ")"            [bracket]

  syntax Cmd ::= "nop" | "accept" | "drop" | "send(" Port "," Fields ")"
               | "alarm(" Exp ")"                                  [strict(1)]
               | "set(" Id "," Exp ")"                             [strict(2)]
               | "newAutomaton(" String "," AutomatonId ")"
               | "step(" AutomatonId "," Exp "," Stmt ")"          [strict(2)]
  syntax Stmt ::= Cmd
                | "cond(" Exp "," Stmt ")"                         [strict(1)]
                | "iter(" Exp "," Stmt ")"                         [strict(1)]
                | "newInterrupt(" Int "," Bool "," Stmt ")"
                | Stmt Stmt                                        [right]
                | "{" Stmt "}"                                     [bracket]

  syntax AutomataDef ::= "AUTOMATA" String AutomataDefTail
  syntax AutomataDefTail ::= "init" "=" AStateId ATransitions | ATransitions
  syntax ATransitions ::= List{ATransition,""}
  syntax ATransition ::= AStateId "-" AEvtId "->" AStateId
  syntax AStateId ::= String
  syntax AEvtId ::= String
  syntax InitSeq ::= "INIT" Stmt
  syntax PrologElt ::= AutomataDef | InitSeq
  syntax Prologues ::= PrologElt | PrologElt Prologues
```

```
47
  syntax Program ::= "PROLOGUE" Prologues "FILTER" Stmt
49
endmodule
51
module GPFPL
53
  imports GPFPL-SYNTAX
55
  configuration
57      <prg color="stmtColor"> $PGM:K </prg>
        <automataKindDefs color="automataColor">
59        <automataKindDef color="automataColor" multiplicity="*">
            <automataKind color="automataColor"> .:String </automataKind>
61          <initialState color="automataColor"> .:AStateId </initialState>
            <transitions color="automataColor"> .Map </transitions>
63        </automataKindDef>
        </automataKindDefs>
65      <br/>
        <filter color="stmtColor"> .:Stmt </filter>
67      <clock color="timeColor"> 0:Time </clock>
        <env color="envColor">
69        <automata color="automataColor">
            <kinds color="automataColor"> .Map </kinds>
71          <states color="envColor"> .Map </states>
          </automata>
73        <vars color="envColor"> .Map </vars>
        </env>
75      <k color="stmtColor"> .K </k>
        <br/>
77      <interrupts color="interruptsColor">
          <nextInterrupts color="interruptsColor"> .List </nextInterrupts>
79        <interrupt multiplicity="*" color="interruptsColor">
            <intTime color="timeColor"> .:Time </intTime>
81          <intCode color="stmtColor"> .:Stmt </intCode>
            <period color="timeColor"> .:Time </period>
83        </interrupt>
        </interrupts>
85      <br/>
        <input color="pktColor">
87        <time color="timeColor"> .:Time </time>
          <port color="pktColor"> .:Port </port>
89        <fields color="pktColor"> .Map:Fields </fields>
        </input>
91      <streams color="streamsColor">
          <in color="streamsColor">
93        <inHead color="pktColor"> .:PktDescr </inHead>
              <inTail stream="stdin" color="streamsColor"> .List </inTail>
95        </in>
          <alarm stream="stdout" color="alarmColor"> .List </alarm>
97        <out stream="stdout" color="streamsColor"> .List </out>
        </streams>
99
syntax KResult ::= ExpVal | Port
101
rule -- I:Int => ~Int I
103 rule I1:Int + I2:Int => I1 +Int I2
    rule I1:Int - I2:Int => I1 -Int I2
105 rule I1:Int * I2:Int => I1 *Int I2
    rule I1:Int / I2:Int => I1 /Int I2 requires I2 =/=Int 0
107
    rule ! B:Bool => notBool B
109 rule B1:Bool & B2:Bool => B1 andBool B2
    rule B1:Bool | B2:Bool => B1 orBool B2
111
    rule I1:Int == I2:Int => I1 =Int I2
113 rule I1:Int < I2:Int => I1 <Int I2
    rule I1:Int > I2:Int => I1 >Int I2
115 rule I1:Int <= I2:Int => I1 <=Int I2
    rule I1:Int >= I2:Int => I1 >=Int I2
117
    rule S1:String + S2:String => S1 +String S2
119 rule S1:String == S2:String => S1 ==String S2
```

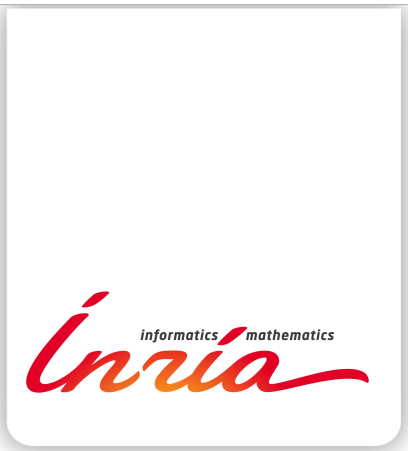
```
121 rule P1:Port == P2:Port => P1 ==Port P2

123   rule
        <k> X:VarId => V ... </k>
125     <vars> ... (X |-> V:ExpVal) ... </vars>

127   rule
        <k> # X:Id => V ... </k>
129     <automata>
          ...
131       <states> ... (X |-> V:AEvtId) ... </states>
          ...
133     </automata>

135   rule
        <k> $ X:Id => ( MF.getValueOfField(X) ) ... </k>
137     <fields> MF:Fields </fields>
        when X in MF
139
      rule
141     <k> _inPort => P ... </k>
        <port> P:Port </port>
143
      rule
145     <prg>PROLOGUE P:Prologues FILTER F:Stmt => P</prg>
        <filter> . => F </filter>
147
      rule
149     <prg> ( P:PrologElt T:Prologues ) => P ~> T </prg>

151   rule
        <prg> AUTOMATA K:String init = S:AStateId T:ATransitions => AUTOMATA K T ...</prg>
153     (. =>
          <automataKindDef>
155     ...
            <automataKind> K </automataKind>
157     <initialState> S </initialState>
        ...
159       </automataKindDef>
        )
161
      syntax ATransitionPremise ::= AStateId AEvtId
163
      rule
165     <prg> AUTOMATA K:String ( (F:AStateId - E:AEvtId -> T:AStateId) TT:ATransitions =>
        TT ) ...</prg>
        <automataKindDef>
167       ...
          <automataKind> K </automataKind>
169       <transitions> _:Map (.Map => (F E):ATransitionPremise |-> T) </transitions>
          ...
171     </automataKindDef>

173   rule
        <prg> AUTOMATA K:String .ATransitions => . ...</prg>
175
      rule
177     <prg> INIT F:Stmt => . ...</prg>
        <k> . => F </k>
179
      syntax Holder ::= "loadInterrupt" | "loadNextPkt"
181
      rule
183     <prg> . </prg>
        <k> . </k>
185     <inHead> . => pktStr2pktDescr(#parse(Input,"PktStr")) </inHead>
        <inTail> ListItem(Input:String) => .List ...</inTail>
187     <input>
          <time> _:K => . </time>
189       <port> _:K => . </port>
          <fields> _:Map => .Map </fields>
191     </input>
```

```
    [structural]

  rule
    <k> . => loadNextPkt </k>
    <inHead> M:PktDescr </inHead>
    <nextInterrupts> .List </nextInterrupts>
    [structural]

  rule
    <k> . => loadNextPkt </k>
    <inHead> P:PktDescr </inHead>
    <nextInterrupts> ListItem(TInt) ... </nextInterrupts>
    when getTimeFromPkt(P) <Time TInt
    [structural]

  rule
    <k> . => loadInterrupt </k>
    <inHead> P:PktDescr </inHead>
    <nextInterrupts> ListItem(TInt) ... </nextInterrupts>
    when notBool ( getTimeFromPkt(P) <Time TInt )
    [structural]

  rule
    <k> loadInterrupt => S </k>
    <nextInterrupts> ListItem(T:Int) => .List ... </nextInterrupts>
    ( <interrupt>
        ...
        <intTime> T </intTime>
        <intCode> S:Stmt </intCode>
        <period> . </period>
    ...
    </interrupt> => . )
    <clock> _ => T </clock>
    [structural]

  rule
    <k> loadInterrupt => S </k>
    <nextInterrupts> ( ListItem(T:Int) => .List ) ( TL:List =>
    insertIntoOrderedTimeList( T +Time P , TL ) ) </nextInterrupts>
    <interrupt>
        ...
      <intTime> T => T +Int P </intTime>
      <intCode> S:Stmt </intCode>
      <period> P:Int </period>
        ...
    </interrupt>
    <clock> _ => T </clock>
    [structural]

  rule
    <filter> F:Stmt </filter>
    <k> loadNextPkt => F </k>
    <inHead> P:PktDescr => . </inHead>
    <clock> _ => getTimeFromPkt(P) </clock>
    <input>
      <time> . => getTimeFromPkt(P) </time>
      <port> . => getPortFromPkt(P) </port>
      <fields> .Map => getFieldsFromPkt(P) </fields>
    </input>
    [structural]

  syntax List ::= "insertIntoOrderedTimeList" "(" Time "," List ")" [function]
  rule insertIntoOrderedTimeList( I:Time , .List ) => ListItem(I)
  rule insertIntoOrderedTimeList( I:Time , ListItem(J) T:List ) => ListItem(I) ListItem(
    J) T when I <Time J
  rule insertIntoOrderedTimeList( I:Time , ListItem(J) T:List ) => ListItem(J)
    insertIntoOrderedTimeList( I , T ) when notBool ( I <Time J )

  rule
    <k> newInterrupt( T:Int , true:Bool , S:Stmt ) => . ...</k>
    <nextInterrupts> IntL:List => insertIntoOrderedTimeList( CT +Time T , IntL ) </
    nextInterrupts>
    <clock> CT:Time </clock>
```

```
    (. =>
      <interrupt>
        <intTime> CT +Time T </intTime>
        <intCode> S </intCode>
        <period> T </period>
      </interrupt>
    )

  rule
    <k> newInterrupt( T:Int , false:Bool , S:Stmt ) => . ...</k>
    <nextInterrupts> IntL:List => insertIntoOrderedTimeList( CT +Time T , IntL ) </
     nextInterrupts>
    <clock> CT:Time </clock>
    (. =>
      <interrupt>
        <intTime> CT +Time T </intTime>
        <intCode> S </intCode>
        <period> . </period>
      </interrupt>
    )

  rule S1:Stmt S2:Stmt => S1 ~> S2    [structural]

  rule cond( true , F:Stmt ) => F
  rule cond( false , _ ) => .K

  rule iter( 0:Int , _ ) => .K
  rule iter( I:Int , F:Stmt ) => F ~> iter( I -Int 1 , F ) requires I >Int 0

  rule
    <k> set( X:Id , V:ExpVal ) => . ... </k>
    <vars> ... X |-> (_ => V ) ... </vars>

  rule
    <k> set( X:Id , V:ExpVal ) ... </k>
    <vars> Rho:Map (.Map => X |-> .) </vars>
    when notBool (X in keys(Rho))
    [structural]

  rule
    <k> newAutomaton( K:String , # X:Id ) => . ... </k>
    <automataKindDef> ...
      <automataKind> K </automataKind>
      <initialState> S </initialState>
    ... </automataKindDef>
    <automata>...
      <kinds> KMap:Map (.Map => X |-> K ) </kinds>
      <states> SMap:Map (.Map => X |-> S) </states>
    ...</automata>
    when notBool (X in keys(KMap) orBool X in keys(SMap))

  rule
    <k> step( # X:Id , E:AEvtId , _:Stmt ) => . ... </k>
    <automataKindDef> ...
      <automataKind> K </automataKind>
      <transitions> ... (F:AStateId E) |-> T ... </transitions>
    ... </automataKindDef>
    <automata>...
      <kinds> ... X |-> K ... </kinds>
      <states> ... X |-> (F:AStateId => T) ... </states>
    ...</automata>

  rule
    <k> step( # X:Id , E:AEvtId , S:Stmt ) => S ... </k>
    <automataKindDef> ...
      <automataKind> K </automataKind>
      <transitions> Transitions:Map </transitions>
    ... </automataKindDef>
    <automata>...
      <kinds> ... X |-> K ... </kinds>
      <states> ... X |-> Fr:AStateId ... </states>
    ...</automata>
    when notBool ( (Fr E) in keys(Transitions) )
```

```
333
  rule
335   <k> alarm( N:String ) => .K ... </k>
      <clock> T:Time </clock>
337   <alarm> ... .List => ListItem("ALARM @ " +String time2Str(T) +String ": " +String N
       +String "\n") </alarm>

339 syntax Cmd ::= InternalCmd

341 syntax InternalCmd ::= "iSend(" Port "," Fields ")" | "iHalt"

343 rule
      <k> iSend( P , F ) => .K ... </k>
345   <clock> T </clock>
      <out> ... .List => ( ListItem( pkt2Str( T , P , F ) +String "\n" ) ) </out>
347   [structural]

349 rule
      <k> ( iHalt ~> _ ) => .K </k>
351   <input>
        <time> _ => . </time>
353     <port> _ => . </port>
        <fields> _ => .Map </fields>
355   </input>
      [structural]
357
  rule
359   <k> send( P , F ) => iSend( P , F ) ... </k>

361 rule
      <k> drop => iHalt ... </k>
363
  rule
365   <k> accept => iSend( oppositePort(P) , MF ) iHalt ... </k>
      <input>
367     ...
        <port> P:Port </port>
369     <fields> MF:Fields </fields>
        ...
371   </input>

373 endmodule
```

Listing: `gpfpl.k3`



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399